Foundations of Machine Learning

Academic Year 2022/2023

Nicola Aggio 880008

Index

1 Information Theory

Information theory was originally proposed by Claude E. **Shannon** (*A mathematical theory of communication*, 1948) and his goal was to **mathematically formalize the concept of information** and, more generally, of **communication**. In this sense, he didn't want to study a specific communication system, but to develop a **general** one.

1.1 Communication system

According to his theory, a communication system can be abstracted as follows:

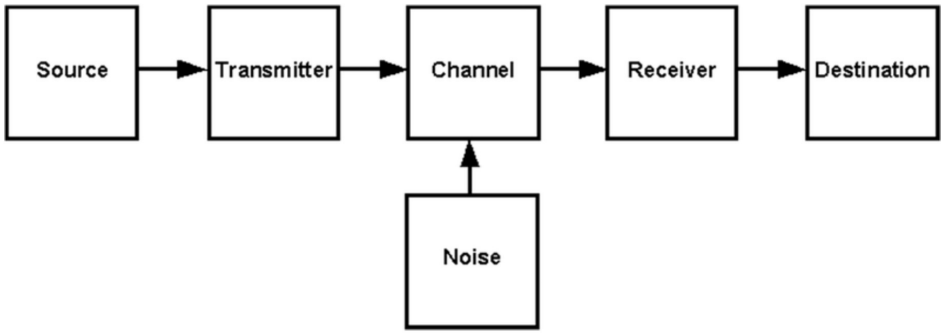


Figure 1: Communication system

This representation is composed by the following components (notice that there could be some variants):

- the **source** and the **destination**. Normally, the source starts the communication by sending some pieces of information, while the destination receives the information. These two entities are not necessarily different;
- a **transmitter**, which translates the message language of the source to the one of the channel;
- a **channel**, which is the medium that allows the transmission of messages between two entities;
- the **noise**, which is an unpredictable phenomenon that can interfere the communication;
- a **receiver**, which translates the message to the language of the destination.

1.2 Information

Another goal of Shannon's work was both to understand the **nature of information** and how to **measure** the amount of information that travels through a channel. In this sense, he distinguished 3 different levels of information:

1. **Symbolic level**, which deals only with the symbols of the message, and it does not consider its semantic (meaning);

2. **Semantic level**, which deals with the meaning of the message, so it tries to understand the semantic. It is very complex;
3. **Pragmatic level**, which studies how the context affects the meaning of a message and what are the intentions of the speaker.

Classical information theory only deals with symbolic level.

1.3 Quantifying the information

Quantifying information refers to the necessity of finding a measure to quantify the **amount of information** that travels inside the **channel**. However, firstly, it is necessary to give a definition of information.
Shannon's intuition was based on the connection between the concept of *information* and *probability* of an event. Suppose we have an event E and its associated probability $P(E)$. We can study the information $I(E)$ provided by the event E as follows:

- If $P(E) = 1$, the amount of information $I(E)$ provided by event E is 0.
- If $P(E) = 0$, the amount of information provided by the event E is ∞.

In this sense, we understand that the concept of *information* is closely related to concepts as **uncertainty** and **surprise**, so in general $I(E)$ can be considered as a function of $P(E)$.

1.3.1 Shannon function

After the premises made so far, we can consider the amount of information $I(E)$ provided by an event E as a function of the probability of the event E: $I(E) = f(P(E))$, which can be represented as follows:

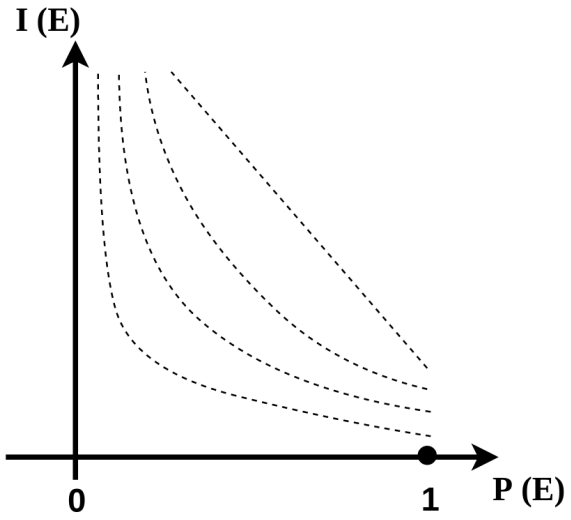


Figure 2: Possible configurations of function $I(E)$.

It is important to notice that we do not know how $I(E)$ behaves in the middle but we can definitely state that it must be **positive** and **monotonically decreasing**. In particular, $I(E)$ is characterized by the following properties:

- $I(E) \geq 0$, with $I(E) = 0 \iff P(E) = 1$;
- $\lim_{P(E)\to 0} I(E) = \infty$;
- $P(E_1) < P(E_2) \implies I(E_1) > I(E_2)$

Shannon proved that there is a **unique function**, the log function, that satisfies these assumptions and respects its definition:

$$I(E) = -\log P(E) = \log \frac{1}{P(E)}$$

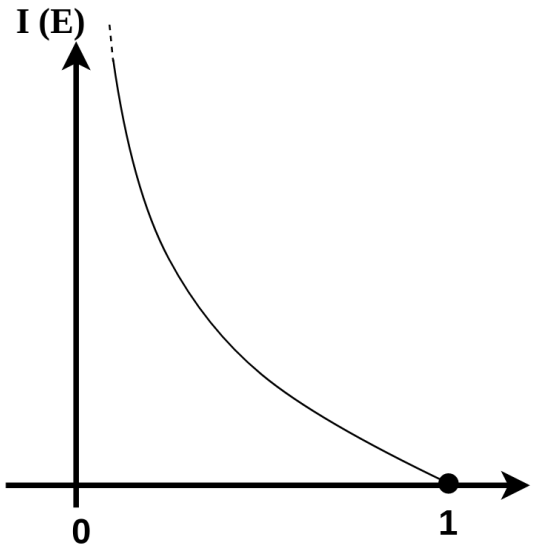


Figure 3: Graphical representation of $I(E) = -\log P(E)$).

NOTE: Why are we dealing with the concept of *events* in a communication system? The act of **producing a symbol** by the source can be considered as an **event**, and, in particular, we can consider the source as a **random variable** (precisely, a **stochastic process**), so that we can measure the amount of information it produces by calculating $-\log P(E)$.

1.3.2 Entropy of a random variable

Given a source it can be interesting to evaluate the amount of information provided by it, and this can be expressed by the concept of **entropy**.
Let X be a random variable with range $\mathcal{X} = \{x_1, x_2, \cdots, x_n\}$ and probability distribution $p(X)$. The **entropy** $H(x)$ represents an average of the amount of information provided by the source, and it is defined as:

$$H(x) = -\sum_{x\in\mathcal{X}} p(x)(x)$$

As we can see, the entropy is nothing but the expected value of X, and it is characterized by the following **properties**:

- The base of logarithm define the measure of information:

$\log_2$	bit
$\log_e$	nat
$\log_{10}$	Hartley

Table 1: Measure of Entropy.

- By convention, $0 \log 0 = 0$, which derives from $\lim_{x \to 0} x \log x = 0$;
- $H_b(x) = (\log_b a) H_a(x)$;
- $H(x) \geq 0$, and $H(x) = 0 \iff x$ has a 0-1 distribution, i.e. a probability distribution with only one point s.t. $p(x) = 1$, and all the others s.t. $p(x) = 0$. In this case, if we know that almost every value of x is equal to 0, then the quantity of information that x provides is close to 0;
- $H(x) \leq \log |\mathcal{X}|$, with $H(x) = \log |\mathcal{X}| \iff x$ has a uniform distribution. In this case, if we have a uniform distribution, each event has the same probability, so we have a maximum uncertainty.

In this sense, we can resume the last two properties as follows:

$$0 \leq H(x) \leq \log(n)$$

Example 1.1. Thinking about the flip of a coin, we can say that the random variable X associated with this event follows the following distribution:

$$X = \begin{cases} \text{head} & \text{w.p. } 0.5 \\ \text{tail} & \text{w.p. } 0.5 \end{cases}$$

Thanks to $H(x)$ it is possible to compute the entropy of this random variable, which means how much information we can get:

$$H(x) = \frac{1}{2} \underbrace{\log 2}_{1} + \frac{1}{2} \underbrace{\log 2}_{1} = 1 \text{ bit}$$

This is a case in which we have a fair coin but if we had an unfair coin (unbalanced), the entropy function would change:

$$X = \begin{cases} x_1 & \text{w.p. } p \\ x_2 & \text{w.p. } 1 - p \end{cases}$$

Now, the *entropy* function $H(x)$ will be equal to:

$$H(x) = -p \cdot \log p - (1 - p) \cdot \log(1 - p) = H(p)$$

A general graphical representation of the entropy function is:

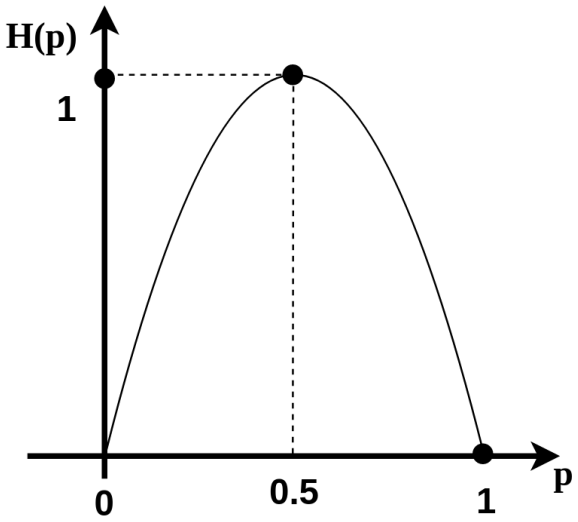


Figure 4: $H(p)$ in relation to p.

As we can see, we have a symmetric and concave function, whose maximum is reached at $p = 0.5$, which represents the point of maximum uncertainty. This is reasonable since there is more surprise for the receiver. Imagine, instead, that $p(x) = 0.8$: the receiver is likely to expect tail, since it has a large probability, hence the surprise will be closer to 0.

1.3.3 Entropy of two random variables

Let's now talk about the entropy of **two random variables**, X and Y, that are defined over the following ranges:

$$\mathcal{X} = \{x_1, \cdots, x_n\} \quad \text{and} \quad \mathcal{Y} = \{y_1, \cdots, y_n\}$$

, while $p(x)$ and $p(y)$ represent the two marginal probability distributions for sources X and Y. Given X and Y, we can consider:

- **Marginal entropy**: $H(X)$ and $H(Y)$;
- **Joint entropy**: $H(X, Y)$, which provides the amount of information given by joining of the two distributions. It is defined as:

  $$H(X, Y) = -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log p(x, y)$$

  , where $p(x, y) = p(y|x)p(x) = p(x|y)p(y)$
- **Conditional entropy**: $H(X|Y = y)$, which provides the amount of information given by X with a fixed value of Y.

  $$H(X|Y = y) = -\sum_{x \in \mathcal{X}} p(x|y) \log p(x|y)$$

  In general, averaging over all possible y's, we obtain the conditional entropy of X given Y:

$$\begin{aligned} H(X|Y) &= -\sum_y p(y) H(X|Y=y) \\ &= -\sum_y p(y) \sum_x p(x|y) \log p(x|y) \\ &= -\sum_x \sum_y p(x|y) p(y) \log p(x|y) \\ &= -\sum_x \sum_y p(x,y) \log p(x|y) \end{aligned}$$

Regarding the average conditional entropy is possible to notice that values can vary in a precise range, in fact:

$$0 \leq H(X|Y) \leq H(X)$$

$\mathbf{H(X|Y) = 0}$ when X is a deterministic function of Y such that:

$$\forall y \in \mathcal{Y} \qquad \exists!\ x \in \mathcal{X} \text{ such that } p(x|y) = 1$$

Choosing the value of y, we know the result of x.

$\mathbf{H(X|Y) = H(X)}$ when X and Y are two independent random variables.

$$p(x,y) = p(x) \cdot p(y)$$

- **Chain rule**, which defines a relationship between joint entropy, marginal entropy and condition entropy. It is defined as:

$$H(X,Y) = H(X) + H(Y|X) = H(Y) + H(X|Y)$$

Indeed:

$$\begin{aligned} H(X,Y) &= -\textstyle\sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} p(x,y) \log p(x,y) \\ &= -\textstyle\sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} p(x,y) \log p(x) p(y|x) \\ &= -\textstyle\sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} p(x,y) \log p(x) - \sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} p(x,y) \log p(y|x) \\ &= -\textstyle\sum_{x\in\mathcal{X}} \left(\sum_{y\in\mathcal{Y}} p(x,y) \right) \log p(x) - \sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} p(x) p(y|x) \log p(y|x) \\ &= -\textstyle\sum_{x\in\mathcal{X}} p(x) \log p(x) - \sum_{x\in\mathcal{X}} p(x) \sum_{y\in\mathcal{Y}} p(y|x) \log p(y|x) \\ &= H(X) - \textstyle\sum_{x\in\mathcal{X}} p(x)(-H(Y|X=x)) \\ &= H(X) + H(Y|X). \end{aligned}$$

This chain rule is derived from the fact that:

$$p(x,y) = p(x) \cdot p(y|x) = p(y) \cdot p(x|y)$$

and by applying the $\log(\cdots)$ function, products become sums.

1.3.4 Entropy of n random variables

Suppose we have n different random variables $X_1, X_2, \cdots, X_n$, then the entropy function is defined as:

$$\begin{aligned} H(X_1, X_2, \cdots, X_n) &= -\sum_{x_1 \in \mathcal{X}_1} \sum_{x_2 \in \mathcal{X}_2} \cdots \sum_{x_n \in \mathcal{X}_n} p(x_1, \cdots, x_n) \cdot \log p(x_1, \cdots, x_n) \\ &= H(X_1) + H(X_2|X_1) + H(X_3|X_1, X_2) + \cdots + H(X_n|X_1, \cdots, X_{n-1}) \\ &= \sum_{i=1}^{n} H(x_i|x_1, \cdots, x_{i-1}) \end{aligned}$$

1.4 Mutual information

In this section we focus on the concept of **mutual information**, which represents a measure of the **mutual dependence** between two variables. More specifically, it quantifies the "amount of information" (in units such as shannons, commonly called bits) obtained about one random variable through observing the other random variable.

1.4.1 Kullback-Leibler divergence

The Kullback-Leibler defines a distance between probability distributions.
Let $\bar{p}$ and $\bar{q}$ be two discrete probability distributions, i.e. two points belonging to the standard simplex $\triangle$, where:

$$\triangle = \{x \in \mathbb{R}^n : \sum_{i=1}^{n} x_i = 1, \qquad x_i \geq 0\}$$

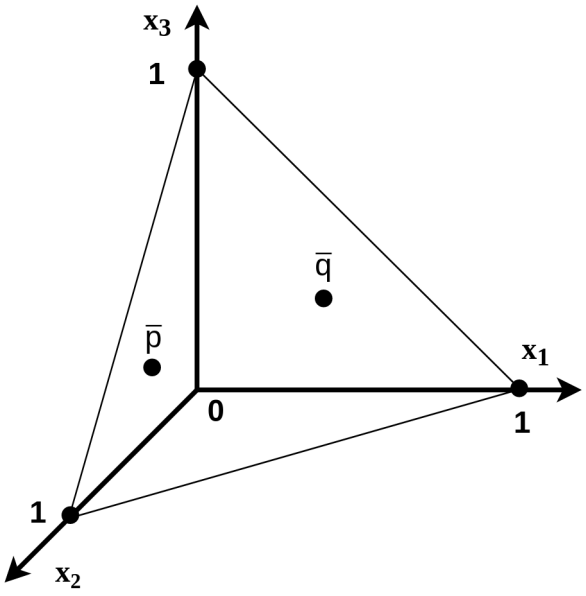


Figure 5: Standard simplex in $\mathbb{R}^3$.

The **Kullback-Leibler divergence** between $\bar{p}$ and $\bar{q}$ is defined as:

$$D(\bar{p}||\bar{q}) = \sum_{i=1}^{n} p_i \log \frac{p_i}{q_i}$$

The KLD is characterized by the following properties:

1. $D(\bar{p}||\bar{q}) \geq 0$, with $D(\bar{p}||\bar{q}) = 0 \iff \bar{p} = \bar{q}$;
2. $D(\bar{p}||\bar{q}) \neq D(\bar{q}||\bar{p})$, i.e. KLD is **not symmetric**;
3. $D(\bar{p}||\bar{r}) \nleq D(\bar{p}||\bar{q}) + D(\bar{q}||\bar{r})$, i.e. KLD does **not** satisfy the **triangular inequality**.

1.4.2 Mutual information

Let X and Y be two discrete r.v., the **mutual information** between them is defined as:

$$\begin{aligned} I(X;Y) &= D\big(p(x,y)||p(x)p(y)\big) \\ &= \sum_x \sum_y p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \end{aligned}$$

Through the computation of mutual information between X and Y, we can have access to different interpretations. One of them is surely understanding if two random variables are **dependent** or not: the **mutual information** of two random variables is **0** if and only if the two random variables are **independent**. Indeed, we have that $D(p(x,y)||p(x)p(y)) = 0$ when x and y are independent, i.e. $p(x,y) = p(x) \cdot p(y)$, meaning that:

$$\log \frac{p(x,y)}{p(x)p(y)} = \log 1 = 0$$

In this sense, the mutual information measures the independence of two r.v.
Moreover, paying attention at the ending part of the computed formula we can understand another important property: the **lower** $\log \frac{p(x,y)}{p(x)p(y)}$ is, the larger the mutual information is, and, consequently, the more **independent** the two random variables are.

1.4.3 Properties of mutual information

Besides providing a measure of the mutual independence between two random variables, the mutual information is also useful when we want to measure how much information travels on the channel.

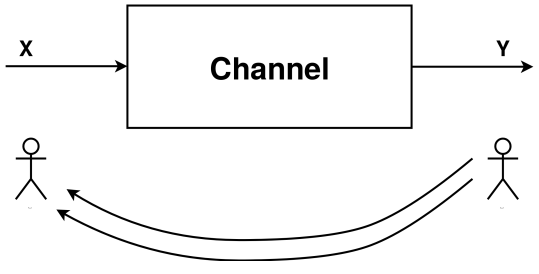


Figure 6: Channel model.

When we deal with the transmission of information, it is important to take into account two different moments: *before* the transmission arrives to the receiver and *after* the transmission arrives to the receiver. Let X be the information transmitted by the source, and Y the information received by the receiver, and suppose that the receiver wants to infer X. Then:

- We denote with $H(X)$ the **uncertainty** of the receiver over X **before** seeing the output from the channel;
- We denote with $H(X|Y)$ the **uncertainty** of the receiver over X **after** looking at Y.

In this sense, the mutual information can be defined as

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$

, and it measures the **amount of information** that travels through a channel.
Some other properties of the mutual information are:

- $I(X;Y) = I(Y;X)$, i.e. the mutual information is **symmetric**;
- $I(X;Y) \geq 0$, with $I(X;Y) = 0 \iff X$and Y are independent;
- $I(X;Y) = H(X) + H(Y) - H(X,Y)$. This derives from the Chain rule, since we know that $H(X|Y) = H(X,Y) - H(Y)$;
- $I(X;X) = H(X)$
- $H(X)$ is related to the **efficiency** of the information channel.
- $I(X;Y)$ is related to the **reliability** of the information channel.

Since $I(X,Y) \geq 0$ we have $H(X) - H(X|Y) \geq 0 \rightarrow H(X) \geq H(X|Y)$ with equality if and only if X and Y are independent.
Picture 7.1 shows the relationship between mutual information and entropy.

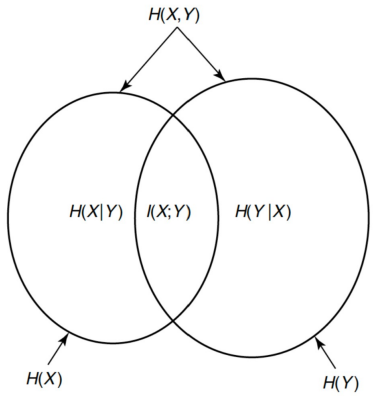


Figure 7: Mutual information and entropy

1.5 Data compression (source coding)

In this section we introduce the notion of **source code**.

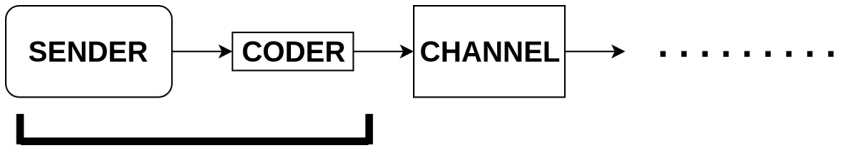


Figure 8: Source coding theorem is focused on the first part of the channel.

As we introduced before, because of the presence of noise inside the channel, source and destination also have the task to ensure **efficiency** and **reliability** of messages. However, ensuring these two properties introduces another problem: **redundancy**. In order to achieve efficiency it is necessary to remove redundancy from the messages, using **compression**; on the other hand, improving reliability requires the receiver to be able to make inferences about the original message, which again can only be achieved through redundancy. As we can see there is an intrinsic **contrast** between the two requirements, hence it is necessary to find a good **trade-off**. A possible solution, for instance, is to send more (possibly an odd number of) bits to codify a single symbol (e.g. `"0"` is encoded as `"000"`). In this way the decoder will more easily be able to spot and fix errors in the received message, increasing reliability as well as redundancy.
Let X be random variable with range $\mathcal{X}$:

$$\mathcal{X} = \{x_1, \cdots, x_n\}$$

and probability distribution

$$p(x) = Pr\{X = x\}$$

We define $\mathcal{D}$ as the channel alphabet and we define a code as a function that executes the following transformation:

$$C : \mathcal{X} \to \mathcal{D}^*$$

, where $\mathcal{D}^*$ represents the set of strings of symbols from the alphabet $\mathcal{D}$. In this sense, a code is a function that maps strings from the source code $\mathcal{X}$ to the channel code $\mathcal{D}$. We denote with $C(x)$ the **codeword** associated to x.
In general, it is clear that not all codes are good ones. However, there are some rules that can help us in order to define a good one:

- It must be as **short** as possible in order to ensure the *efficiency* of transmission.
- There must not be presence of codes that are prefixes of other ones: this leads problems with *efficiency* because receiver must wait other bits in order to understand if sender has sent letter `"b"` or `"d"`.

It is also possible to have an efficient code that is useless, because receiver can't understand what is transmitted in the channel. An example is the following table in which **ambiguity** appears.

$\mathcal{X}$	$\mathcal{D}^*$
1	0
2	0
3	0
4	0

Table 2: Ambiguous code.

1.5.1 Classes of codes

We can define the following classes of codes:

1. **Non-singular codes**, in which all the codewords are distinct (i.e. the coding function C is injective). Notice that non-singular codes can be not uniquely decodable, as represented below;

$\mathcal{X}$	$\mathcal{D}^*$
1	0
2	010
3	01
4	10

Table 3: Example of non-singular but not uniquely decodable code.

   A code like this can generate a message such as 010, in which a decoder can generate more than one decoded message. For instance, it could generate 2, 14 or 31.. As we can see, even if we are in an optimal situation without noise, the receiver can't understand the message it received.

2. **Uniquely decodable**, in which any encoded string must have a unique decoding. An example is provided here:

$\mathcal{X}$	$\mathcal{D}^*$
1	10
2	00
3	11
4	110

Table 4: Example of uniquely decodable but not instantaneous code.

3. **Instantaneous code** or **Prefix code**, in which no codeword is a prefix of any other. An example is provided here:

$\mathcal{X}$	$\mathcal{D}^*$
1	0
2	10
3	110
4	111

Table 5: Example of instantaneous (or prefix) code.

Notice that a uniquely decodable code is also non-singular (it is not always true the opposite), and an instantaneous/prefix code is also uniquely decodable: a scheme of the classes of codes is provided here:

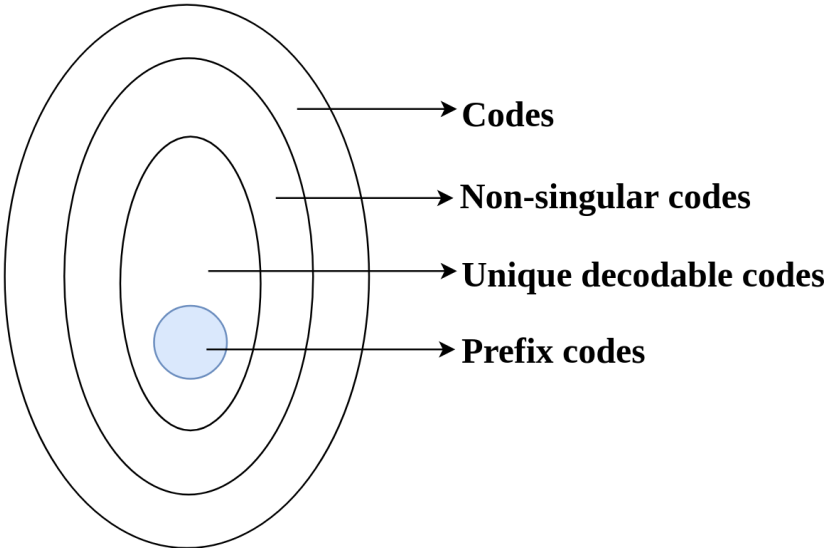


Figure 9: Codes classification.

1.6 Quantifying efficiency

While we define a code, it is also useful to consider the efficiency given by it. If we consider a code like this:

$\mathcal{X}$	$\mathcal{D}_1^*$	$\mathcal{D}_2^*$
a	0	0
b	10	10001
c	110	1100110
d	111	1110010

Table 6: Non-efficient code.

, we can see immediately that the codewords use too many bits to codify the original symbol of the source. So we could consider the **length of codewords** as a **possible metric** to quantify efficiency of a code, but we will find out in the next example that this metric is not the best one.

Example 1.2. Consider the following example:

$$\mathcal{X} = \{a, b, c, d\} \qquad \mathcal{D} = \{0, 1\}$$

with the following probabilities:

$$p(a) = \frac{1}{2} \qquad p(b) = \frac{1}{4} \qquad p(c) = \frac{1}{8} \qquad p(d) = \frac{1}{8}$$

We can define two possible codes to convert symbols from $\mathcal{X}$ to $\mathcal{D}$.

$\mathcal{X}$	$\mathcal{D}_1^*$	$\mathcal{D}_2^*$
a	0	111
b	10	110
c	110	10
d	111	0

Table 7: Comparing codes.

The second code is the reversed version of the first one. If we consider only the length of the codewords we can say that in terms of efficiency they are equal, but this is not true. When we talk about efficiency, it is necessary to consider the probability distribution of the symbols. In this example the symbol a is very likely to appear, while symbol d is very unlikely to be found. In other words, while the second code maps a to a long codeword and d to a short one, the first code does the opposite, hence proving to be more efficient since it uses, on average, fewer bits.
We can thus define the **length of a code** with the following formula:

$$L(C) = \sum_{x \in \mathcal{X}} p(x) l(x)$$

, i.e. it is the average length of the codewords, where $l(x)$ is the length of the codeword associated to the symbol x.

Example 1.3. Let X be a random variable with following range:

$$\mathcal{X} = \{1, 2, 3, 4\} \qquad \mathcal{D} = \{0, 1\}$$

and the following probability distribution:

$$p(X = 1) = \frac{1}{2} \qquad p(X = 2) = \frac{1}{4} \qquad p(X = 3) = \frac{1}{8} \qquad p(X = 4) = \frac{1}{8}$$

Then:

$$L(C) = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3 = 1.75$$

Notice that in this case $H(X) = 1.75 = L(C)$.

Example 1.4. Let X be a random variable with following range:

$$\mathcal{X} = \{1, 2, 3\} \qquad \mathcal{D} = \{0, 1\}$$

and the following probability distribution:

$$p(X = 1) = \frac{1}{3} \qquad p(X = 2) = \frac{1}{3} \qquad p(X = 3) = \frac{1}{3}$$

Then:

$$L(C) = \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 2 + \frac{1}{3} \cdot 3 = 1.66$$

However, in this case $H(X) = \log_2 3 = 1.58$ bits.

There is a theorem which states that entropy is a lower bound for $L(C)$, considering a **noise free channel**.

Theorem 1.5 (Shannon's theorem)**.** *Let X be a random variable (source), with range $\mathcal{X}$ and probability distribution $p(x)$. Let $\mathcal{D}$ be the channel's alphabet, $C : \mathcal{X} \to \mathcal{D}^*$ an instantaneous code for X and $D = |\mathcal{D}|$.*
Then, if the channel is noise free:

$$L(C) \geq H_D(X)$$

where D is the base of the logarithm.
Moreover:

$$L(C) = H_D(x) \iff \forall x \in \mathcal{X} : \quad l(x) = -\log_D p(x) = \log_D \frac{1}{p(x)}$$

When the probability distribution $p(x)$ has the property:

$$\log_D \frac{1}{p(x)} \in \mathbb{N} \setminus \{0\}$$

the probability distribution $p(x)$ is called **D-adic**. Thus, the equality in the theorem is reached if and only if the probability distribution is *D-adic*.

Example 1.6.

$$\mathcal{X} = \{a, b, c, d\} \qquad \mathcal{D} = \{0, 1\}$$

with the following probabilities:

$$p(a) = \frac{1}{2} \qquad p(b) = \frac{1}{4} \qquad p(c) = \frac{1}{8} \qquad p(d) = \frac{1}{8}$$

$$p(a) = \frac{1}{2} \quad \log_2 2 = 1$$
$$p(b) = \frac{1}{4} \quad \log_2 4 = 2$$
$$p(c) = \frac{1}{8} \quad \log_2 8 = 3$$
$$p(d) = \frac{1}{8} \quad \log_2 8 = 3$$

1.7 Huffman coding

The **Huffman coding** is a **greedy algorithm** used to build/define the **best code** in a lossless data compression.
The algorithm proceeds as follows:

1. Take the two least probable symbols in the alphabet. These two symbols will be given the longest codewords, which will have equal length, and differ only in the last digit;

2. Combine these two symbols into a single one, and repeat.

Example 1.7. Huffman coding with $\mathcal{X} = \{x_1, \ldots, x_6\}$ and $p(x) = (0.4, 0.3, 0.1, 0.1, 0.06, 0.04)$.

$\mathcal{X}$	$p(x)$	c_5	$p(x)^1$	c_4	$p(x)^2$	c_3	$p(x)^3$	c_2	$p(x)^4$	c_1
x_1	0.4	1	0.4	1	0.4	1	0.4	1	0.6	0
x_2	0.3	0	0.3	00	0.3	00	*0.3	00	0.4	1
x_3	0.1	011	*0.1	011	*0.2	010	*0.3	01		
x_4	0.1	010	*0.1	0100	*0.1	011				
x_5	*0.06	01010	0.1	0101						
x_6	*0.04	01011								

Table 8: Comparing codes.

Theorem 1.8 (Optimality of Huffman codes)**.** *Huffman codes are optimal (and instantaneous).*

Theorem 1.9. *If $C_{Huffman}$ is a Huffman D-ary code for a random variable X, then:*

$$H_D(X) \leq L(C_{Huffman}) < H_D(X) + 1$$

Example 1.10. Let X be a r.v. with range $\mathcal{X} = \{x_1, x_2\}$ and $p(x_1) = \frac{2}{3}$ and $p(x_2) = \frac{1}{3}$. Then, the optimal (Huffman) code is :

$$x_1 \to 0$$

$$x_2 \to 1$$

In this case, the length of the code is 1, and we can't apparently do better than this, but let's consider blocks of two consecutive symbols. Assuming statistical independence, we have $p(x_1, x_1) = \frac{4}{9}$, $p(x_1, x_2) = \frac{2}{9}$, $p(x_2, x_1) = \frac{2}{9}$, $p(x_2, x_2) = \frac{1}{9}$. An optimal (Huffman) code for this new source is :

$$x_1x_1 \to 0$$

$$x_1x_2 \to 10$$

$$x_2x_1 \to 110$$

$$x_2x_2 \to 111$$

Now, the expected codeword length per input symbol is 0.944, which is less than before! Clearly, increasing the input size, we obtain better and better codes. Shannon's theorem states that this process will converge towards the entropy of X, in this case $H(X) = 0.91830$

1.8 Channel

In the previous sections, we covered the topic of coding. However, two different definitions of codes can be indicated:

- **Lossless codes**, i.e. codes that don't lose any information from the initial source. It is possible to reconstruct completely the original message;
- **Lossy codes**, i.e. codes that lose information. After compression some information is lost (jpg, ...).

Two meaningful concepts are **source coding**, which is related to *efficiency*, and **channel coding**, that is, instead, related to *reliability*. We will now focus exactly on this last topic.

1.8.1 Definition of channel

A channel can be formally defined as a triplet:

$$\mathcal{C} = (\mathcal{X}, p(y|x), \mathcal{Y})$$

where:

- $\mathcal{X}$ is the **input alphabet**.
- $p(y|x)$ is the **channel's probability distribution**. It represents the probability that symbol x is translated into y. It is the most important component of the channel and it is defined as:

$$p(y|x) = Pr(Y = y|X = x)$$

- $\mathcal{Y}$ is the **output alphabet**.

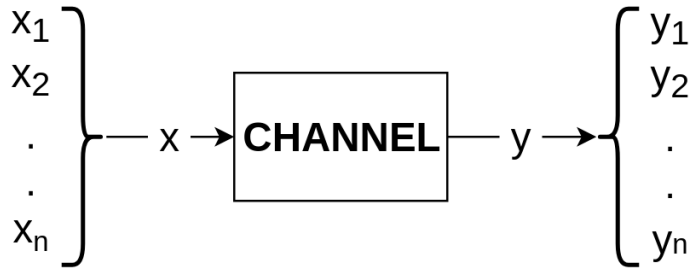


Figure 10: Channel definition.

1.8.2 Capacity of a channel

Let us consider a channel $\mathcal{C}$, we want to analyze the mutual information $I(X;Y)$. We have defined the mutual information as:

$$I(X;Y) = H(X) - H(X|Y)$$

where:

- $H(X)$ is considered as $f(p(x))$ function of the probability distribution of the source.
- $H(X|Y) = -\sum_x \sum_y p(x,y) \log p(x|y)$, where:
  - $p(x,y) = p(x)p(y|x)$ in which $p(x)$ is related to the *source* and $p(y|x)$ is related to the *channel*. Generally, it is possible to say that $p(x,y)$ depends both on the source and on the channel.
  - From Bayes Theorem we also know that

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)}$$

in which again $p(y|x)$ is related to the *channel* and instead $p(x)$ is related to the *source*.

$$p(y) = \sum_x p(x,y) = \sum_x \underbrace{p(x)}_{source} \underbrace{p(y|x)}_{channel}$$

The mutual information that travels on the channel depends both on the source and on the channel.

$$I(X;Y) = f(p(x), p(y|x)) = f(source, channel)$$

where:

- $p(x)$ is the "a priori" probability.
- $p(y|x)$ is the "a posteriori" probability.

This means that we can't use mutual information to give capacity information of the channel since $I(x,y)$ is not only dependent on the channel itself, but also on the source. After all these premises, we are able to define the **capacity of channel** as:

$$C = \max_{p(x)} I(X;Y)$$

The **capacity** represents the maximum value of mutual information given by all the possible probability distributions of the source.

Example 1.11. We compute the capacity of the following Binary Symmetric Channel:

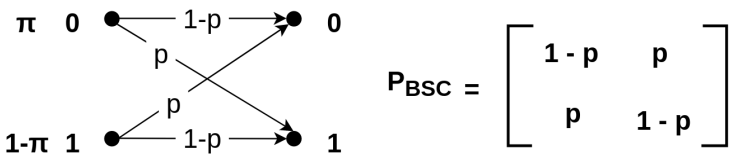


Figure 11: Binary symmetric channel.

We know that $P\{X = 0\} = \pi$ and $P\{X = 1\} = 1 - \pi$ (with $\pi \in [0,1]$).
What could happen in the different extreme cases is explained in the following lines:

- $p = 0$, we have that the channel is **perfect** and its capacity is 1. This represent the ideal case in which the channel is not affected by noise and receiver knows that the messages are correct. No loosing of information.

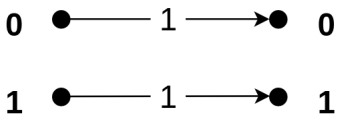


Figure 12: Channel representation when $p = 0$.

- $p = 1$, we have that the channel is again **perfect**. Receiver always knows how to reconstruct the right message. The message is always modified during the transmission on the channel. The channel's capacity is again 1.

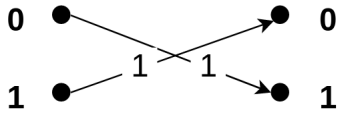


Figure 13: Channel representation when $p = 1$.

- $p = 0.5$ is the worst scenario for the receiver. We can consider the starting message X and the final message Y as independent messages.

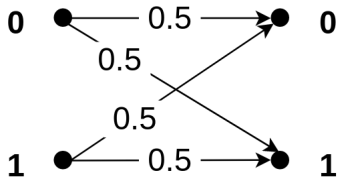


Figure 14: Channel representation when $p = 0.5$.

Here there is a strong assumption: the **receiver knows the probability distribution of the channel.** The best case for the receiver is to know both the channel distribution and the sender distribution, but this is not always feasible. Probabilities of the channel, in fact, can be specified only after several trials and they are not fixed, as they could change during the time.

1.9 Reliability

When we have a noisy channel, the simplest way to improve reliability is exploit the **repetition of the code**, i.e. add redundancy to the code, in order to reduce the probability of errors during the communication.
Notice that in this case the receiver decides how to decode the redundant message by taking, for example, a majority vote. Assume that a channel is BSC with error probability $p = 10^{-2}$. Then, using a repetition code of length 3, we reduce this error probability to $3 \cdot 10^{-4}$, but at the same time we reduced the transmission rate to $\frac{1}{3}$.

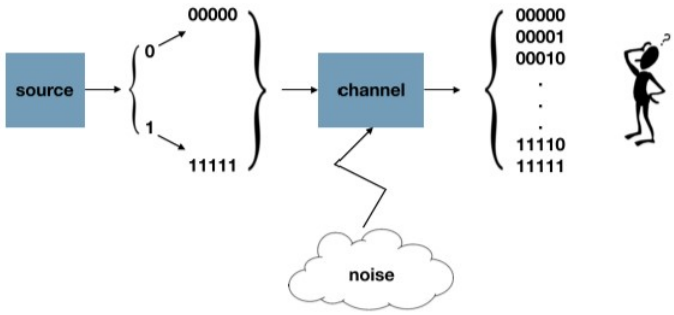


Figure 15: Adding redundancy in a noisy channel

If we use a repetition code of length 5, we reduce the probability of error to approximately 10^{-5}, reducing the transmission rate to $\frac{1}{5}$.
What if we now consider $n \to \infty$? Let's define P_e as the error probability. When $n \to \infty$ we have that $P_e \to 0$ but $R \to 0$, meaning that a more reliable code produces a slower communication system. It is necessary to find a right trade-off to maintain a good level of reliability and speed.

1.10 Channel coding theorem - Shannon's 2nd theorem

Theorem 1.12 (Channel coding theorem)**.** *Let $\mathcal{C}$ be a channel with capacity C.*

- *If $R < C$, there exists a sequence of codes with transmission rate R such that:*

$$P_e \underset{n\to\infty}{\to} 0$$

- *Conversely, if a sequence of codes transmitting at rate R has a probability error approaching to zero (i.e. $P_e \to 0$), then $R \leq C$.*

In this sense, this theorem states that we can send a message **at a finite rate** through a noisy channel with error probability **as small as we want**, provided that we transmit at a rate smaller than the capacity. Thus, the **capacity** of the channel is an **upper bound** for R if we want to define a reliable code.

2 Neural Networks

A Neural Network is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the new structure of the information processing system. It is composed of a large number of highly interconnected processing elements (**neurons**) working in unison to solve specific problems. A NN is configured for a specific application, such as pattern recognition or data classification, through a learning process.
Two key **features** distinguish neural networks from any other sort of computing developed:

- **Neural networks are adaptive or trainable**. Neural networks are not so much programmed as they are trained with data. The more data they are fed, the more accurate or complete is their response;
- **Neural network are naturally massively parallel**. This suggests they should be able to make decisions at high-speed and be fault tolerant (information is stored in a distributed fashion).

2.1 Biological digression

As we introduced before, the functioning of a Neural Network is highly inspired by the way in which the neural system and the brain process the information. The biological neural network is composed by the following elements:

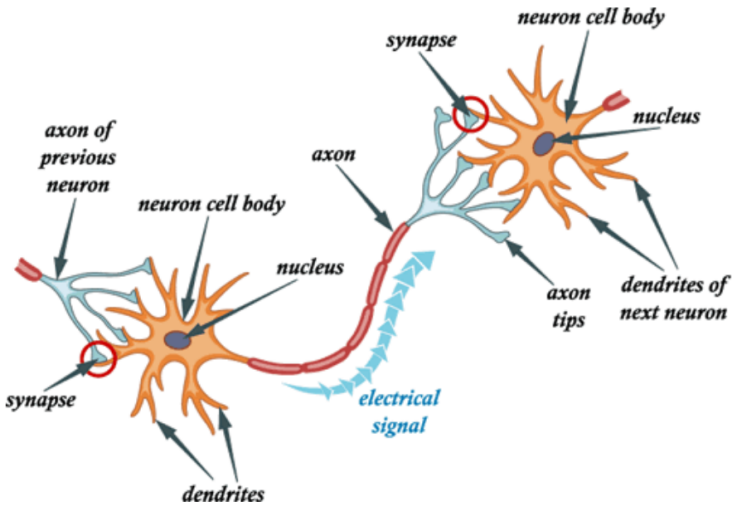


Figure 16: Neural dynamics.

- **Cell body (Soma)**: 5-10 microns in diameter, it represents the *computational unit*;
- **Axon**: it represents the output mechanism for a neuron. A single axon may be connected to thousands of branches and cells;
- **Dendrites**: they receive incoming signals from other nerve axons via synapse;
- **Synapses**: they represents the junctions ("connecting points") between neurons, i.e. the point in which a neuron's axon passes the information to the "next" neuron's dendrite.

The **transmission of signal** in the cerebral cortex is a complex process:

$$\text{Electrical} \rightarrow \text{Chemical} \rightarrow \text{Electrical}$$

Simplifying:

1. The cellular body performs a "*weighted sum*" of the incoming signals.
2. If the result exceeds a certain threshold value, then it produces an "action potential" which is sent down the axon (cell has "fired"), otherwise it remains in a rest state.
3. When the electrical signal reaches the synapse, it allows the "neuro-transmitter" (chemical) to be released. This combines with the "receptors" in the post-synaptic membrane.
4. The post-synaptic receptors provoke the diffusion of an electrical signal in the post-synaptic neuron.

In summary, computations by neuron are performed thanks to a combination of electrical and chemical processes.
A very important concept of the neural system is the so-called **synaptic efficacy**, which can be defined as the amount of electricity that enters into the post-synaptic neuron, compared to the action potential of the pre-synaptic neuron. The **learning step** takes place by modifying the synaptic efficacy. In general, two different types of synapses are present:

- **Excitatory**, which favor the generation of action potential in the post-synaptic neuron, i.e. they tend to increase the energy;
- **Inhibitory**, which hinder the generation of action potential, so they de-amplify the signal.

2.2 The McCulloch and Pitts Model (1943)

Neural network simulations appear to be a recent development; however, this field was established before the advent of computers and the first model, which was created in 1943, is the **McCulloch and Pitts Model**.
The McCulloch-Pitts (MP) neuron is a simple process unit modeled as a binary threshold unit.

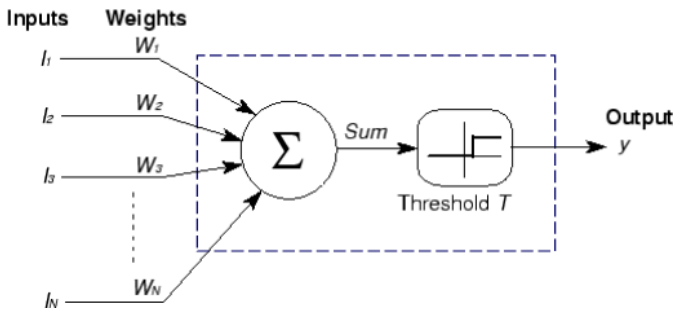


Figure 17: McCulloch and Pitts neuron representation.

Input $= x = \sum_j w_j I_j$.

Output $= g(x) = \begin{cases} 0 & \text{if } x < T \\ 1 & \text{if } x \geq T \end{cases}$

In this sense, we can rewrite the output as:

$$y = g\left(\sum_j w_j I_j - T\right)$$

NOTE:

- the MP neuron fires if the input $x = \sum_j w_j I_j$ exceeds a certain threshold T, called **claiming parameter**;
- the function $g(.)$ is also called **activation function** or **unit step function**, and it is a non linear function;
- the weight w_{ij} represents the strength of the synapse between neuron i and neuron j.

2.2.1 Properties

By properly combining MP neurons, it is possible to simulate the behavior of any boolean circuit:

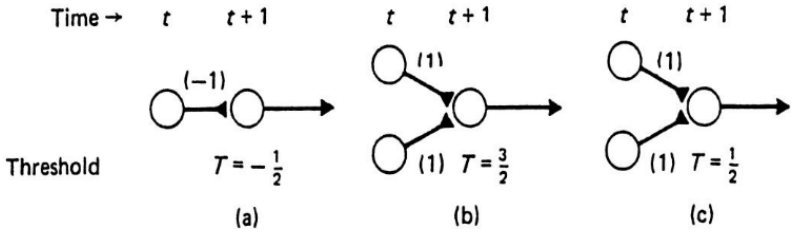


Figure 18: Three elementary logical operations (a) **negation**, (b) **and**, (c) **or**. In each diagram the states of the neurons on the left are at time t and those on the right at time $t + 1$.

Notice that it is not possible to build a NN for the *XOR* operator using a single neuron.

2.3 Network topologies and Architectures

There are different network topologies and the main differences are highlighted below.

Feed-forward only: allow signals to travel one way only: from input to output, so it has connections only in one direction. This topology forms a direct acyclic graph, and its outputs are deterministic functions of the input	**Recurrent networks**: can have signals traveling in both directions by introducing loops in the network. Feedback networks are powerful and can get extremely complicated, since their output depends on the initial state, which in turn depends on previous outputs.
Fully connected: each neuron of a layer is connected to every neuron in the previous layer, and each connection has it's own weight.	**Sparsely connected**: has fewer links than the possible maximum number of links within that network.
Single layer: every neuron connects directly from the network's input to its output	**Multi-layer**: it has one or more layer of hidden neurons that are not connected to the output.

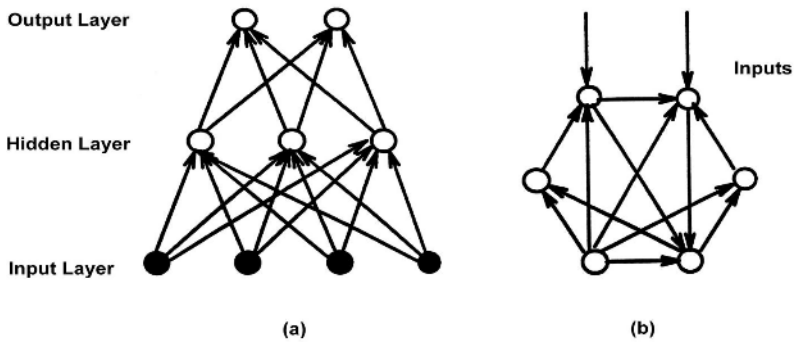


Figure 19: *(a)* feedforward network - *(b)* feedback network

In general, there are problems which are more suited to be solved with a *feedforward NN* than a *recurrent NN* or vice-versa, for example a task of image classification, i.e. assigning a label to an input image, can be easily solved with a *feedforward NN*, whereas a task of image captioning, i.e. provide a verbal description of the input image, is more suitable to be solved with a *recurrent NN*, since the length of the output is not known in advance.

2.4 Classification problems

Given:

1. a set of **features** $\{f_1, f_2, \ldots, f_n\}$, which represents the attributes that describe our objects;
2. a set of **classes** $\{c_1, \ldots, c_m\}$, which represents the categories in which the objects are divided

, the **goal** of the **classification** problem is to classify the **objects** according to their **features**. The geometric interpretation of this task is to represent the features in the space and try to separate the classes by finding the closest objects.

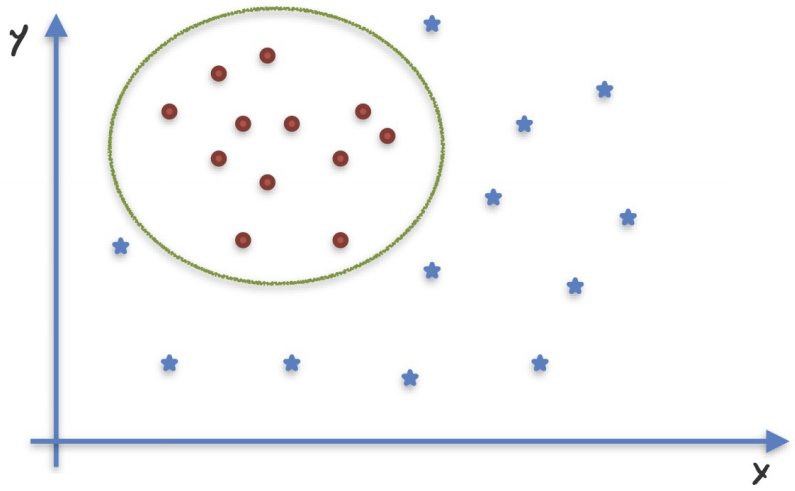


Figure 20: Classification problem

2.4.1 Neural networks for classification

A neural networks can be used as a classification device and it can be configured as follow:

- the *input* of the network are the object's **features** to classify.
- the *output*, returned by the network, is the **class** predicted for the input object.

Before going on we assume that features are numbers, and remember that we cannot map categorical features into numbers, since we would introduce an order, which would lead to mistakes. For instance: red = 0, blue = 1, green = 2 would not be a correct coding since we would be imposing a non-existent order between colors. We propose a simple model of neural network:

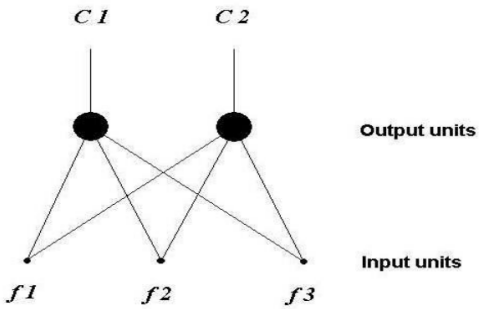


Figure 21: Example of NN with 3 features input and 2 class labels as output.

The application of a learning algorithm of a network configuration consists in finding the best configuration of weights of the incoming connection and the threshold. In these classification problems we can get rid of the **thresholds** (also called biases) associated to neurons by adding an extra input **permanently clamped at -1**. By doing so, **thresholds become weights** and can be adaptively adjusted during learning phase, otherwise

we would have to manually tune the right threshold. In this way, the output y of the network becomes:

$$y = g\left(\sum_{i=1}^{n+1} w_i x_i\right)$$

, where $x_{n+1} = -1$.

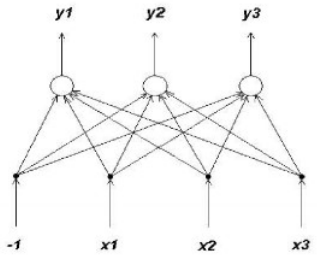


Figure 22: We can remove thresholds by adding an input neuron clamped at -1

2.4.2 The Perceptron

The perceptron is a **linear classifier** consisting of a **single layer of MP neurons** connected in a **feedforward** way, i.e. it is a **single-layer feedforward NN**. A simple perceptron can only solve linearly separable problems, and this makes the model very limited in terms of computational power.

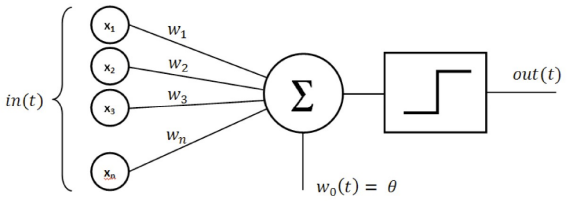


Figure 23: Representation of a perceptron

The perceptron is characterized by the following properties:

- The output is $-1/+1$, while using the MP neurons we had $0/1$;
- It is capable of learning from examples;
- From a geometrical point of view, the perceptron identifies a linear boundary between two classes:

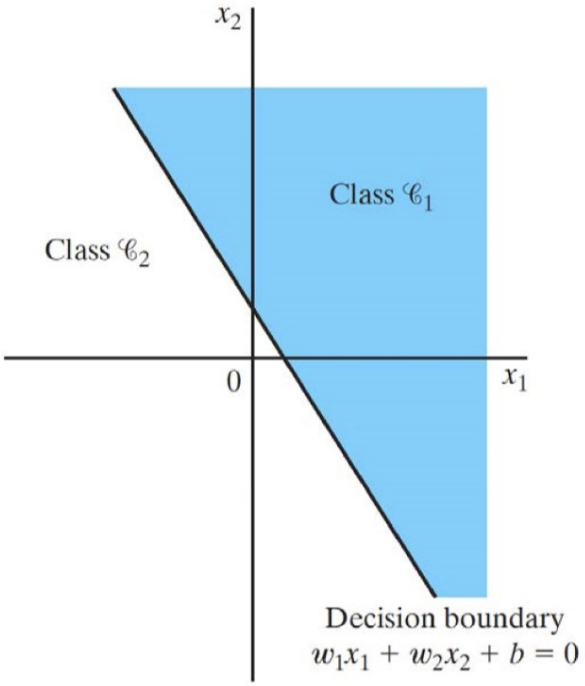


Figure 24: Geometry interpretation of the perceptron

In this case, the hyperplane (line) $w_1x_1 + w_2x_2 - b$ represents a linear decision boundary for a two-dimensional, two-classes classification problem. If $w_1x_1 + w_2x_2 - b > 0$ then the predicted class is 1, otherwise is 0. Obviously, if we change the parameters w_1, w_2 and b, we get a different boundary.

The Perceptron Learning Algorithm. The goal of this algorithm is to find the best weights and parameters in order to separate the objects of the classes. It is an iterative algorithm characterized by the following variables and parameters:

- $x(n) \in \mathbb{R}^{m+1}$, the **input vectors**. They are $(m+1)$-by-1 vectors $= [-1, x_1(n), x_2(n), \ldots, x_m(n)]^T$ Objects are described with m features and -1 as the threshold, which is the reason why the input vectors have size $m + 1$;
- $w(n) \in \mathbb{R}^{m+1}$, the **weights vectors**. They are $(m+1)$-by-1 vectors $= [b, w_1(n), w_2(n), \ldots, w_m(n)]^T$
- b, the **bias**;
- $y(n)$, the **actual response** of the NN (quantized). $y(n) \in \{+1, -1\}$;
- $d(n)$, the **desired response** from the dataset. $d(n) \in \{+1, -1\}$;
- η, the **learning-rate parameter**, a positive constant strictly smaller than 1. It affects the convergence of the learning algorithm.

The algorithm works as follows:

1. **Initialization.** Set $w(0) = 0$ and then perform the following computations for time-step $n = 1, 2, \ldots$.

2. **Activation.** At time-step n, activate the perceptron by applying continuous-valued input vector $x(n)$ and desired response $d(n)$ (both picked from the training set).
3. **Computation of the actual response.** Compute the actual response of the perceptron as:
$$y(n) = sgn[w^T(n)x(n)] \qquad w^T x = \sum_i w_i x_i$$
where $sgn(\cdot)$ is the signum function (+1 if the argument is greater or equal to 0, 0 otherwise). Notice that, so far, no learning phase has taken place.
4. **Update of the weight vector.** Update the weight vector of the perceptron to obtain:
$$w(n+1) = w(n) + \eta[d(n) - y(n)]x(n)$$
where
$$d(n) = \begin{cases} +1 & \text{if } x(n) \text{ belongs to class } \varphi_1 \\ -1 & \text{if } x(n) \text{ belongs to class } \varphi_2 \end{cases}$$
This is where the learning phase takes place: in case of wrong classification, the current configuration will change modifying $d(n) - y(n)$, if the network classifies $x(n)$ correctly, then there is no learning as $d(n) - y(n) = 0$. The convergence of the algorithm is ensured by $x(n)$. Note that $x(n)$ is included to be sure that the new prediction is correct, but the updated weights could fail to classify another input vector (even an already seen one).
5. **Continuation.** Increment time step n by one and go back to step 2.

In practice, this algorithm is trying to find the best possible straight line (or hyperplane) which separates the "good" examples from the "bad" ones. The decision boundary is described such that $w_1x_1 + w_2x_2 = 0$ and, in general, each point is classified according to the following conditions: $w_1x_1 + w_2x_2 \geq 0$ and $w_1x_1 + w_2x_2 < 0$

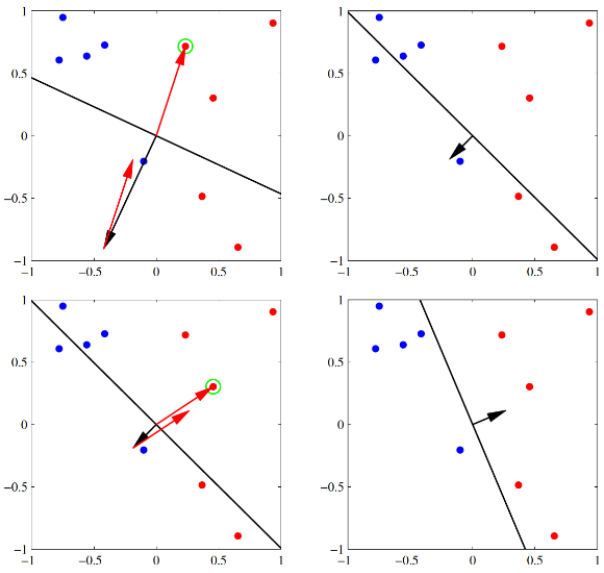


Figure 25: Perceptron learning algorithm procedure.

From a geometrical point of view, finding w means finding the line (or hyperplane) which separates the two regions. As we can see from the previous image, some examples can be misclassified. For each weights update, a specific error is corrected but other mistakes in classification may arise because of the changes in the weights. By iteratively adjusting the weights, a final and "stable" solution can be reached.
It is possible to define a **decision region** as an area in which all the samples of one class fall. A classification problem is said to be **linearly separable** if the decision regions can be separated by an hyperplane. These concepts allow us to introduce one important **limitation of perceptrons**: they can only solve linearly separable problems.

The Perceptron Convergence Theorem. This theorem was formulated by Rosenblatt in 1960. It states the following: if the training set is **linearly separable**, the perceptron learning algorithm **always converges** to a consistent hypothesis after a **finite** number of epochs, i.e. an entire presentation of the dataset, for any $\eta > 0$ (Note: nothing is said with respect to the number of steps).
If the training set is **not linearly separable**, after a certain number of epochs the weights will start oscillating. In this situation some points will surely be misclassified. The learning algorithm will never converge to a stable configuration, due to the fact that the perceptron can't solve non-linear classification problems.

2.5 Multi-Layer Feed-forward Neural Networks

In the following image it is possible to find some properties of the different structures that can create a neural network.

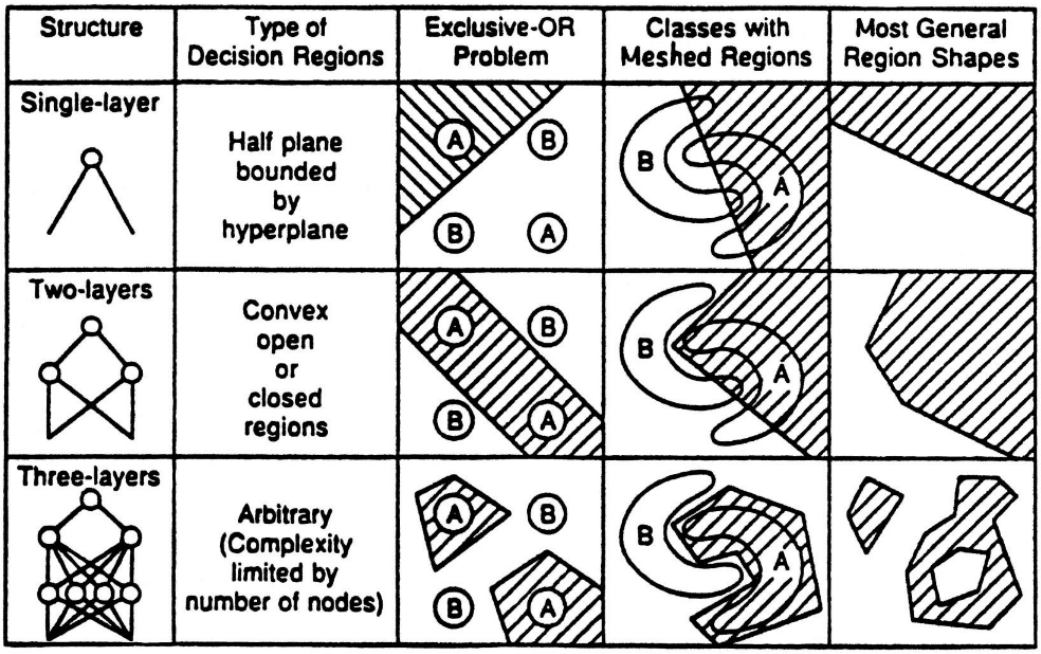


Figure 26: A view of the Role of Units

As we said before, the limit of the perceptron algorithm is that it can only deal with

linearly separable problems. In order to overcome these limitations, the introduction of **multi-layer feed-forward networks** allows us to improve our networks through the addition of **hidden layers** between the input and the output layer. This allows us to reach a good approximation on our classification problem: it is possible to notice that a network with just one hidden layer can represent any Boolean function, including XOR. One property of such networks is the **universal approximation power**, which states that a 2-layers network (one input layer, one hidden layer) can approximate any smooth function (valid for regression problems), provided that the hidden layer is large enough.

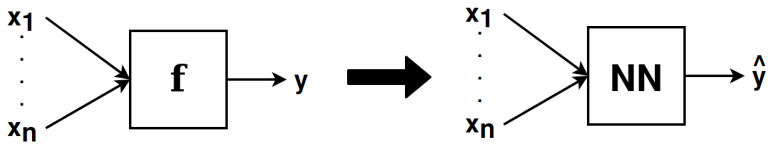


Figure 27: Neural Network function model.

Continuous-valued units. The usage of continuous-valued units allows us to introduce calculus and derivative procedures that will give us several advantages.
The **activation function** of continuous-valued units is **sigmoid** (or logistic), which means that neurons can now fire with different intensities, not only 0/1. In this sense, their output is not boolean, but it belongs to the continuous range between 0 and 1. These values are often interpreted as probabilities.

$$g \to \sigma(x) = \frac{1}{1 + e^{-f(x)}} \in (0, 1)$$

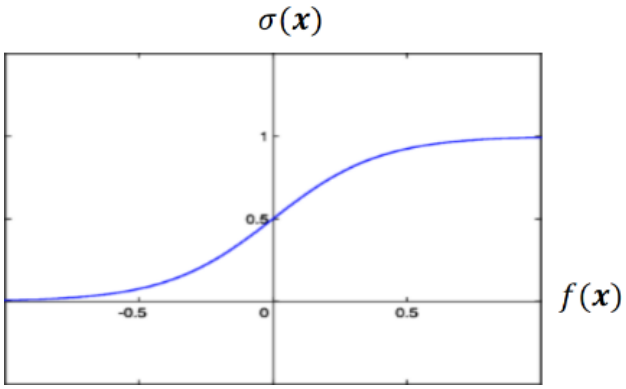


Figure 28: Sigmoid function.

Thus, continuous-valued discriminant functions allow us to have some measure of **confidence** about the **prediction**. We will see later on that they are used for **finding optimal solutions through the gradient descent technique**. In particular, when the $g(z)$ is close to zero (less confidence about the prediction) the gradient will be greater then the situation in which $|g(x)| >> 0$ (more confidence, weights could remain stable).
Another possible activation function is the **hyperbolic tangent** ($tanh$), which is a rescaling of the logistic sigmoid such that its output range is from -1 and 1.

$$g \to \sigma(x) = \frac{e^{f(x)} - e^{-f(x)}}{e^{f(x)} + e^{-f(x)}} \in (-1, 1)$$

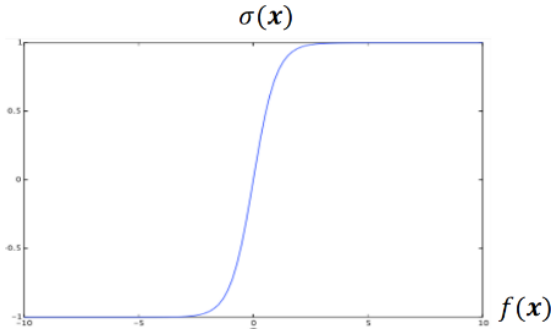


Figure 29: Hyperbolic tangent function.

2.6 Back-propagation learning algorithm

In this section we introduce the **backpropagation**, an algorithm for learning the weights in a feed-forward multilayers neural network.
Let $\mathcal{L} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ be a **training set**, where $X = \{x_1, x_2, .., x_n\}$ represents the network **input** vector and $Y = \{y_1, y_2, .., y_n\}$ represents the **desired** network **output** vector. The algorithm is based on **gradient descent** method, and it can be seen as a greedy algorithm with infinite possible solutions, in which at each step the best solution is chosen. The algorithm is guaranteed to converge only to a local maximum and during the execution we move along the gradient through a predetermined step-size. This property is given by the fact that this algorithm is just an approximation based on a greedy solution.

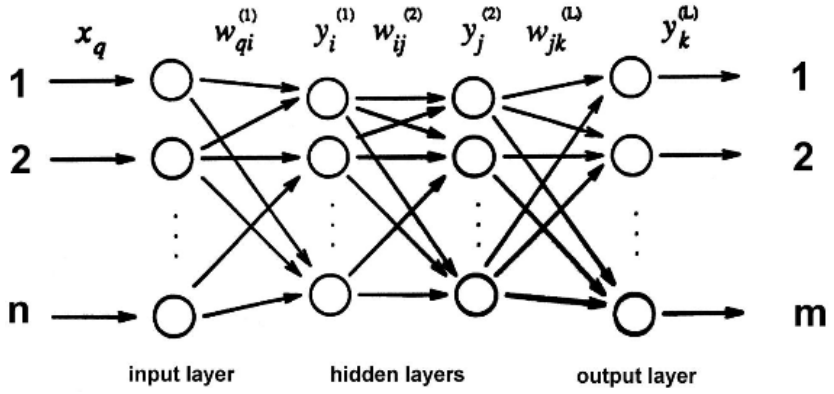


Figure 30: Back-propagation schema. W_{ij}^{l} represents the weight on connection between the i^{th} unit in layer $(l-1)$ to j^{th} unit in layer l.

Supervised learning. Supervised learning algorithms require the presence of **previous knowledge** that makes it possible to provide the **right answers** to the input "questions". Technically this means that we need a **training set** of the form:

$$\mathcal{L} = \left\{\left(x^1, y^1\right), \ldots, \left(x^N, y^N\right)\right\}$$

where:

$$x^{\mu}(\mu = 1, \ldots, N) \text{ is the network } \textbf{input} \text{ vector}$$

$y^{\mu}(\mu = 1, \ldots, N)$ is the **desired** network **output** vector

The learning (or **training**) phase consists in determining a configuration of **weights** such that the **network output** should be **as close as possible** to the **desired output** as many times as possible, for all the examples in the training set. Normally, this amounts to minimizing an **error function** such as the **MSE** (Mean Squared Error) function:

$$E(w) = \frac{1}{2} \sum_{\mu} \sum_{k} (y_k^{\mu} - O_k^{\mu}(w))^2$$

where $O_k^{\mu}(w)$ is the **output** provided by the **output unit** k when the network is given example μ as **input**. In other words, $O_k^{\mu}(w)$ represents the prediction of the k^{th} unit when the input is μ.
The loss function computes the difference between the network output and the expected output and of course, our aim is to **minimize the loss function** (i.e. solve $min_w E(w)$), which means **finding the set of weights that minimizes the function**: if the network is performing well, then $E(w$ is close to zero.
In order to minimize the error function E, we can use the classic **gradient descent** algorithm. The gradient gives information about the **best path to follow** in order to **maximize/minimize our objective function**. Without any kind of direction information it would be extremely complex to find the solution since we would have to search along an infinite number of directions in a continuous domain. Gradient descent is a very well-known **greedy algorithm**. It is not guaranteed to find a globally optimal solution, but it works well in finding **local optima**.
More specifically, the gradient descent updates the a point by moving along the the direction given by the gradient by a finite step. In the case of the back-propagation algorithm, we have that:

$$w_{ji}^{\text{NEW}} \leftarrow w_{ji}^{\text{OLD}} - \eta \frac{\partial E}{\partial w_{ji}}$$

or

$$\Delta w_{ji} \leftarrow -\eta \frac{\partial E}{\partial w_{ji}^{\text{OLD}}}$$

, where :

- $\Delta w_{ji} = w_{ij}^{\text{NEW}} - w_{ji}^{\text{OLD}}$
- w_{ji}^{NEW} represent the weights between unit i and unit j, and they are updated using the gradient of the loss function E computed w.r.t. w_{ji}^{OLD};
- η represents the **learning rate**, i.e. how much the weights are updated at each iteration of the gradient descent method. We will discuss about the possible values of this parameter later in the section;

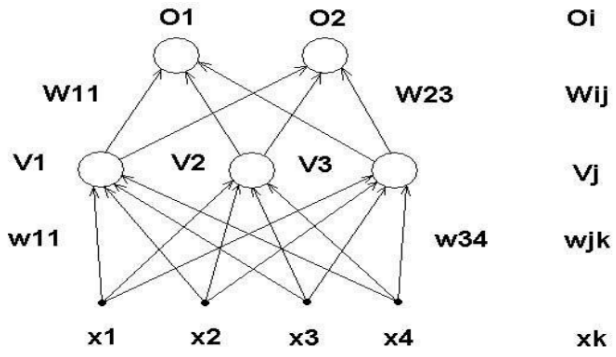
Now we need to define a method for computing the partial derivatives of the loss function efficiently.
To compute the partial derivatives we use the **error back propagation** algorithm, which consists of two stages:

- **Forward pass**: the **input** of the network is **propagated** layer after layer in **forward** direction.
- **Backward pass**: the **error** (e.g. MSE) made by the network is **propagated backward** and **weights** are **updated** properly. Thus, the partial derivatives of the errors are computed, and the weights are updated following the gradient descent strategy we defined above. An important thing to underline is that in order to update each weight, we only need a **local information**, not a global information of all the network.

Intuitively, we could say that we determine the gradient direction and then we move in the opposite direction since we want to minimize the error.

Notation. Before understanding how the weights are updated, it is important to introduce some notation that will be used later:

- x_k represents an input neuron of the network;
- w_{jk} represents a weight between the input x_k and the neuron V_j in the middle layer;
- V_j represents a neuron in the hidden layer;
- W_{ij} represents a weight between a neuron in the hidden layer V_j and the output neuron O_i;
- O_i represents a output neuron.



Given a pattern μ, an hidden unit j receives a net input

$$h_j^\mu = \sum_k w_{jk} x_k^\mu$$

and produces as output:

$$V_j^\mu = g\left(h_j^\mu\right) = g\left(\sum_k w_{jk} x_k^\mu\right)$$

The output of a neuron in the output layer is defined as:

$$O_i^\mu = g(\sum_k W_{ik} \cdot V_k^\mu)$$

Updating hidden-to-output weights. The **updating rules** of the back-propagation algorithm are nothing but a **long series of chain rules**. For this reason, the objective function is not linear, hence the output is highly non linear. Indeed, the output is a chain of products of non-linear functions.

$$\Delta W_{ij} = -\eta \frac{\partial E}{\partial W_{ij}} \quad \text{Replace the error function with its actual expression}$$

$$= -\eta \frac{\partial}{\partial W_{ij}} \left[\frac{1}{2} \sum_{\mu} \sum_{k} (y_k^\mu - O_k^\mu)^2 \right] \quad \text{First application of the chain rule}$$

$$= \eta \sum_{\mu} \sum_{k} (y_k^\mu - O_k^\mu) \frac{\partial O_k^\mu}{\partial W_{ij}} \quad \text{The sum over } k \text{ disappears because the partial derivative is different from 0 only when } k = i$$

$$= \eta \sum_{\mu} (y_i^\mu - O_i^\mu) \frac{\partial O_i^\mu}{\partial W_{ij}} \quad \text{Again we apply the chain rule. Partial derivative can be written as } W_{ij} \to h_i^\mu \to g'(h_i^\mu)$$

$$= \eta \sum_{\mu} (y_i^\mu - O_i^\mu) g'(h_i^\mu) V_j^\mu \quad \text{That is } \frac{\partial h_i^\mu}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}} \left(\sum_l W_{il} V_l \right) = V_j \frac{\partial W_{ij}}{\partial W_{ij}} = V_j$$

$$= \eta \sum_{\mu} \delta_i^\mu V_j^\mu \qquad where: \quad \delta_i^\mu = (y_i^\mu - O_i^\mu) g'(h_i^\mu)$$

In $\eta \sum_\mu \delta_i^\mu V_j^\mu$, component δ_i^μ represents the error made by the i-th neuron, whilst V_j^μ is the output of the neuron.

Updating input-to-hidden weights.

$$\Delta w_{jk} = -\eta \frac{\partial E}{\partial w_{jk}}$$

$$= \eta \sum_{\mu} \sum_{i} (y_i^\mu - O_i^\mu) \frac{\partial O_i^\mu}{\partial w_{jk}} \quad \text{Chain rule}$$

$$= \eta \sum_{\mu} \sum_{i} (y_i^\mu - O_i^\mu) g'(h_i^\mu) \frac{\partial h_i^\mu}{\partial w_{jk}} \quad \text{Chain rule}$$

We have that the partial derivative is:

$$\begin{aligned}
\frac{\partial h_i^\mu}{\partial w_{jk}} &= \sum_l w_{il} \frac{\partial V_l^\mu}{\partial w_{jk}} \\
&= w_{ij} \frac{\partial V_j^\mu}{\partial w_{jk}} \\
&= w_{ij} \frac{\partial g\left(h_j^\mu\right)}{\partial w_{jk}} \\
&= w_{ij} g'\left(h_j^\mu\right) \frac{\partial h_j^\mu}{\partial w_{jk}} \\
&= w_{ij} g'\left(h_j^\mu\right) \frac{\partial}{\partial w_{jk}} \sum_m w_{jm} x_m^\mu \\
&= w_{ij} g'\left(h_j^\mu\right) x_k^\mu
\end{aligned}$$

Hence coming back to the original equation we have that:

$$\begin{aligned}
\Delta w_{jk} &= \eta \sum_{\mu,i} \left(y_i^\mu - O_i^\mu\right) g'\left(h_i^\mu\right) w_{ij}\ g'\left(h_j^\mu\right) x_k^\mu \\
&= \eta \sum_{\mu,i} \delta_i^\mu w_{ij} g'\left(h_j^\mu\right) x_k^\mu \\
&= \eta \sum_\mu \hat{\delta}_j^\mu x_k^\mu \qquad\qquad where: \quad \hat{\delta}_j^\mu = g'\left(h_j^\mu\right) \sum_i \delta_i^\mu w_{ij}
\end{aligned} \tag{1}$$

$\sum_i \delta_i^\mu W_{ij}$ is the average error performed by the output layer and i is the reference to the i-th neuron.
In the following image it is possible to understand the error back-propagation. The black lines correspond to the forwarded signals, while the red lines indicate the error that is back-propagated.

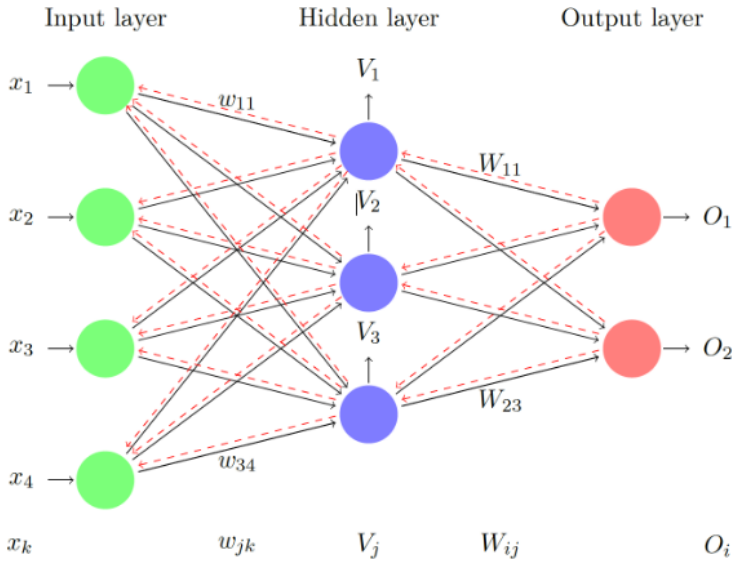


Figure 31: Error Back-propagation representation.

Locality of back-propagation. Another important aspect is the **locality** of the back-propagation algorithm.

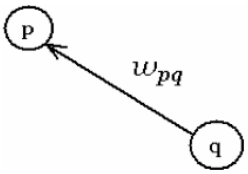


Figure 32: Locality of back-propagation

There exist two different ways of implementing a back-propagation algorithm: off-line and on-line. In the **off-line** way we compute the gradient exactly, so in order to update a single weight in the network, we need to present all the examples of the training set. This procedure is not generally used because it is computationally expensive.

$$\Delta\omega_{pq} = \eta \sum_{\mu} \delta_p^{\mu} V_q^{\mu} \qquad \text{off-line}$$

$$\delta_p^{\mu} = \text{Error} \quad V_q^{\mu} = \text{Output of neurons}$$

In the **on-line** way some noise is introduced, in the sense that the weights are updated at any presentation of an example of the training set.

$$\Delta\omega_{pq} = \eta \delta_p^{\mu} V_q^{\mu} \qquad \text{on-line}$$

A possible compromise between the two techniques is called **stochastic gradient descent**, in which the first technique is used with a randomly selected subset of the training set: in this sense, the gradient changes according to the random choice of the subset. This solution represents a very good method, since on the one hand modern datasets are huge, so updating the weights after the presentation of all the training examples is very expensive, and on the other the randomness through which the subset is selected may help in finding a global maximum/minimum.

The Back-Propagation algorithm. In this implementation we will consider the **on-line** approach. Suppose we have a network with M layers, and let

- V_i^m be the output of the i-th unit of layer m;
- w_{ij}^m be the weight on the connection between j-th neuron of layer $m-1$ and i-th neuron in layer m.

The general structure of the algorithm is the following:

1. Initialize the weight to (small) random values. The choice of small values is made to avoid the derivative of the logistic function being all zeros;
2. Choose a pattern $\bar{x}^{\mu}$ and apply it to the input layer ($m = 0$), i.e.:

$$V_k^0 = x_k^{\mu} \qquad \forall k$$

3. Propagate the signal forward (**Forward pass**):

$$V_i^m = g\left(h_i^m\right) = g\left(\sum_j w_{ij} V_j^{m-1}\right)$$

4. Compute the δ's for the output layer (**Backward pass**):

$$\delta_i^m = g'\left(h_i^m\right)\left(y_i^m - V_i^m\right)$$

5. Compute the δ's for all preceding layers (for each neuron):

$$\delta_i^{m-1} = g'\left(h_i^{m-1}\right) \sum_j w_{ji}^m \delta_j^m$$

6. Update connection weights according to the error, with learning rate η:

$$w_{ij}^{NEW} = w_{ij}^{OLD} + \Delta w_{ij} \qquad \text{where} \qquad \Delta w_{ij} = \eta \delta_i^m V_j^{m-1}$$

7. Go back to step 2 until convergence, i.e. until $\Delta w_{ij} = 0$. In real implementation we claim $||\Delta w_{ij}||^2 < \epsilon$, or to stop after a finite number of epochs.

One of the most challenging problems is the choice of the **learning rate** η. When η is **small** the algorithm will converge but in a very **slow** way; on the other hand, when it is too **big**, there will be an **oscillating problem** that won't bring the algorithm to the convergence.
In the following image we can see the difference in convergence between the same algorithm run with different values of η. From left to right, they are $0.02, 0.0476, 0.049, 0.0505$.

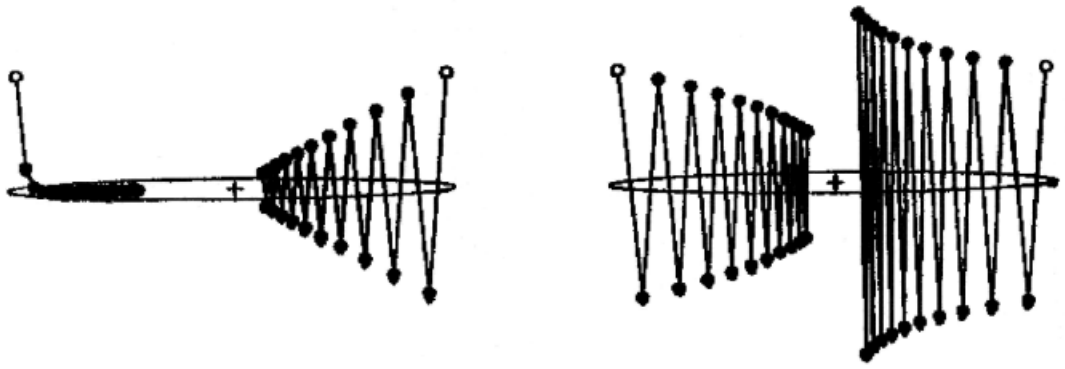


Figure 33: The Role of the Learning Rate. The minimum is at the + and the ellipse shows a constant error contour.

We can observe that the first case is too slow in reaching the minimum, while in the second and in the third case the oscillation becomes smaller and smaller. Finally, in the last case, the algorithm won't converge, as the oscillation becomes larger and larger.

One possible remedy to this problem is in introducing the **momentum term**, a simple **heuristic** that can help in finding a good value for η. This improvement has the advantage

of introducing a correction that is based on the step at time $t-1$, while the original algorithm produces a correction which is only related to the current point.

$$\Delta\omega_{pq}(t+1) = -\eta\frac{\partial E}{\partial w_{pq}} + \underbrace{\alpha\Delta w_{pq}(t)}_{\text{momentum}}$$

- If $\alpha = 0$ we come back to the original formula $\Delta = f(w^T)$
- Otherwise $\Delta = f(w^T, w^{t-1})$

The momentum term introduces **dependency** on the previous step. The obvious disadvantage is the need to **set two parameters instead of one**. On the other hand the momentum term **allows** us to **use large values** of η avoiding the introduction of the oscillatory phenomena. The application of the momentum term can be seen in the following image.

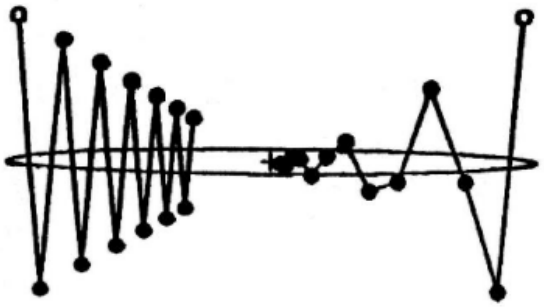


Figure 34: Momentum term application.

Both trajectories use $\eta = 0.0476$ that is the best value of learning rate in the absence of momentum. In the left example, no momentum is applied ($\alpha = 0$), while in the right one we have $\alpha = 0.5$. The application of momentum, hence, brings a clear improvement in convergence.

The problem of local minima. One of the toughest disadvantages to overcome is the fact that back-propagation cannot avoid the **problem of local minima**., i.e. the fact that the solution provided by the gradient descent approach is not global but only local.

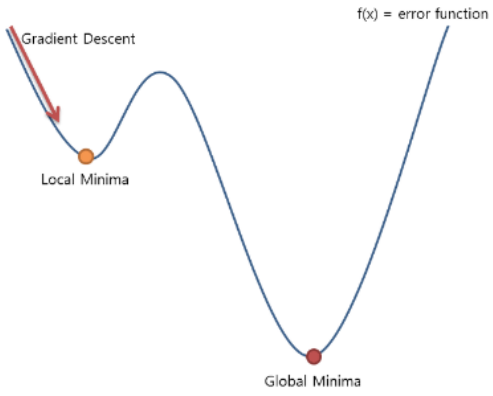


Figure 35: The problem of local minima.

For this reason the **choice** of **initial weights** is of uttermost importance. If the weights are too large, the non-linearities tend to saturate since the beginning of the learning process.
A common heuristic for the choice of the initial weights is:

$$w_{ij} \simeq 1/\sqrt{K_i}$$

, where k_i is the number of units that feed unit i (the "fan-in" of i)

NETtalk. **NETtalk** represents an example of application of the Back-propagation algorithm. This model is composed by a Neural Network and a speech synthesizer, and its goal is to pronounce the string that is provided as input to the network. In this case, each of the letter in input is represented using one-hot-encoding: there are 203 input units encoding the letters and 1 hidden layer with 80 units, while the output units encode English phonemes.
The model was trained by 1024 words in context, and it was able to produce intelligible speech after 10 training epochs, resulting to be functionally equivalent to DECtalk, an expert system, with the difference that this model does not require any linguistic knowledge.

2.7 Theoretical and practical questions

- How many layers are needed for a given task? If the problem is linearly separable, than 1 layer is enough (Perceptron), otherwise we also noticed that 2 layers are enough to solve any problem, provided that the layers are large enough, which sometimes is unfeasible.
- How many units per layer should we use?
- To what extent does representation matter?
- What do we mean by generalization?
- What can we expect from a network as far as generalization is concerned?

    - **Generalization:** it can be defined as the **performance** of the network on **data not** included in the **training set**. One of the major **advantages** of neural nets is their ability to **generalize**, i.e. to classify data (belonging to the same class as the learning data) that it has never seen before. To reach the best generalization, the dataset should be split into three parts: **training set**, **validation set** and **test set**.
      The learning should be stopped when the minimum of the validation set error is reached. At this point the net should be generalizing in the best possible way. When learning is not stopped, "overtraining" occurs and the performance of the net on the dataset as a whole decreases, despite the fact that the error on the training data still gets smaller. In this case, we say the network is *overfitting*. As a matter of fact, after finishing the learning phase the net should be evaluated on the third data set, the test set.
    - Size of the training set: how large should a training set be for "good" generalization?

- Size of the network: too many weights in a network may result in poor generalization.

2.8 Model evaluation

When we talk about model evaluation, it is important to understand how the performance of a model can be evaluated. An intuitive idea is that the **lower** the **error** generated by the model is, the **better** the **model** is. In this sense, it is possible to discern between two different types of errors:

- The **true error** (denoted as $error_{\mathcal{D}}(h)$) of hypothesis h with respect to target function f and distribution $\mathcal{D}$, is defined as the probability that h will misclassify an instance drawn at random according to $\mathcal{D}$.

  $$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}} [f(x) \neq h(x)]$$

  In other words, the true error measures the probability of making a mistake in real life. However, notice that for computing this quantity it is necessary to have knowledge of the probability distribution, which is usually unknown;
- The **sample error** (denoted as $error_s(h)$) of hypothesis h with respect to target function f and data sample S is:

  $$error_S(h) \equiv \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x))$$

  , where n is the number of examples in S, and the quantity $\delta(f(x), h(x))$ is 1 if $f(x) \neq h(x)$, and 0 otherwise. The sample error comes from the idea of estimating the probability distribution of the data from the samples available.

The **true error** is **unknown** (and will remain so forever), while in order to compute the sample error, it is possible to split the dataset, keeping a percentage for the training set and a percentage for the test set. Once the model is trained, it is evaluated over the test set, i.e. by measuring its error on unseen data. Clearly, a good choice of the training set and the test set is crucial.

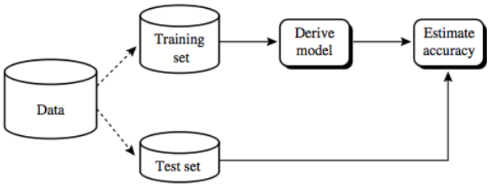


Figure 36: Training set vs Test set.

Cross-Validation Cross-validation is a technique that avoids the chance that the choice of the test and training sets affect the model evaluation. This technique is based on iteratively splitting the dataset into a number of training and test folds such that at each iteration the model is both trained and evaluated considering different examples of the dataset.

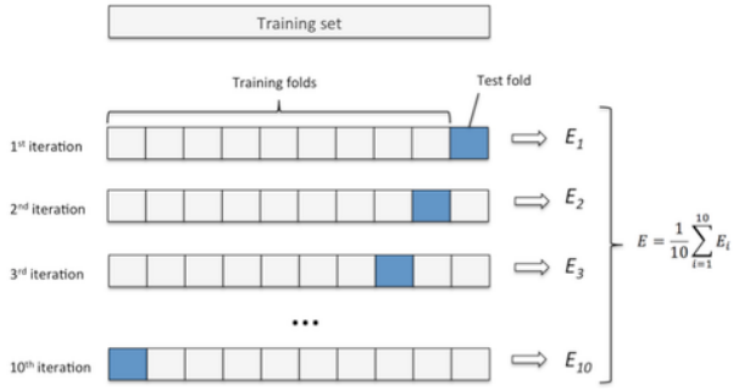


Figure 37: Cross validation example using **leave-one-out** technique, i.e. the size of the test fold is 1.

Clearly, the **advantage** of this method is that it provides a very accurate measure of the error of the model, but on the other hand the drawback is its complexity in time when the datasets are large.

Overfitting Even though cross-validation can give us a good evaluation of the model, it is possible that this score could be affected of **overfitting**.

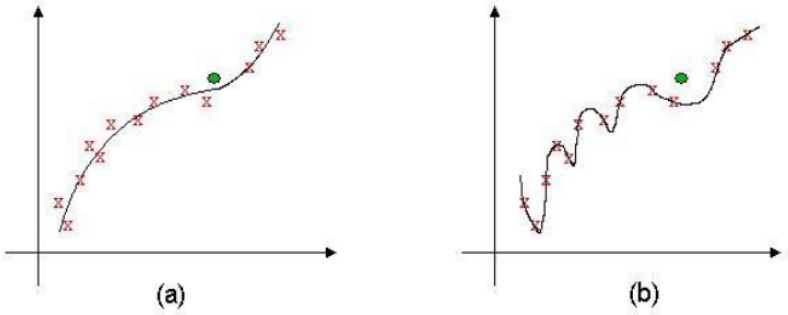


Figure 38: Example of overfitting.

In the image **(a)**, we notice a good fit to noisy data and the model seems to be using fewer parameters to capture the general behavior. In image **(b)**, instead, an overfitted model can be seen: the fit is perfect on the training set, but it is likely to be poor on the test set represented by the circle, i.e. it is characterized by a poor generalization power. **Occam's razor** intuitively explains why the simplest model is to be preferred.
The different steps in the model definition are reached through the separation of the starting set in:

- **Training set:** to train the learning algorithm;
- **Validation set:** to stop the learning algorithm. In particular, it is used to evaluate whether the model is facing overfitting or not by measuring the error on unseen data. In this sense, its functioning is very similar to the test set, but the usage is different, since in this case its goal is to measure the minimum error in order to decide the stopping time;

- **Test set:** to evaluate the performance of the learning algorithm.

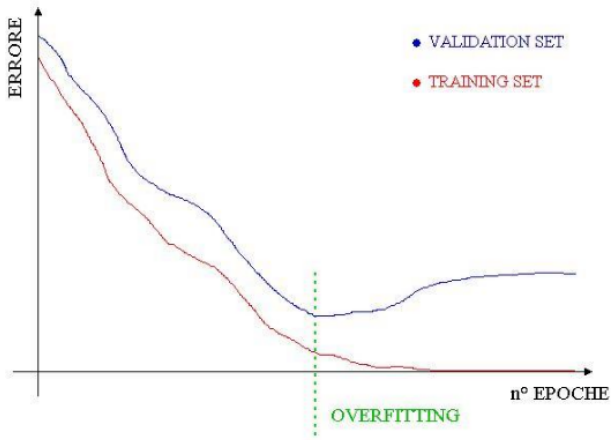


Figure 39: Early stopping.

Epochs of a machine learning algorithm represent the steps taken by the algorithm. Starting from the green line (overfitting line), the model will start to overfit on the training data. Notice also that the global optimum consists in an overfitted model, with the error function approaching zero. The learning algorithm is stopped when the fit starts deteriorating.

Size of a Neural Network The size of a NN, i.e. the number of hidden neurons, affects both its functional capabilities and its generalization performance.

- A **big network** leads to poor generalization performance and to the phenomenon of overfitting;
- A **small network** could not be able to model the desired input/output mapping and for this reason it would not actually learn anything. This phenomenon is called underfitting.

In general, it is hard to tell when the algorithm should stop because it is impossible to foresee if increasing the number of neurons would significantly decrease the error. This problem is known as the **horizon effect**. A common strategy is called **growing**: the procedure starts with one neuron and trains it. Going on, it will iteratively add neurons until satisfying results are obtained.

The pruning approach. This approach is based on the idea of **training an over-dimensioned network** and then **removing redundant nodes** and connections (**offline pruning**). The idea of pruning is to start with a large number of neurons and reduce it as much as possible. Pruning reduces the final complexity of the classifier, hence improving the network's predictive accuracy by avoiding overfitting. Notice that the pruning could also take place during the training phase (**online pruning**), but in this case it would lead to some changes to the learning algorithm, so it is not a common choice.
The most important **advantages** of this technique are:

- Arbitrarily complex decision regions;
- The training is faster;
- Independence of the training algorithm.

The pruning approach has some **disadvantages**: in a network with the "perfect" number of neurons, back-propagation can fail because of the limited number of degrees of freedom. Usually it is not a good idea to apply learning algorithms on networks with the exact number of neurons due to the limitation imposed by the degrees of freedom. Having more degrees of freedom implies higher chances and more different ways to reach the desired goal.

The first issue is how to **choose the unit to be removed**: ideally, we would like to remove the neuron with lowest residuals. However, this is computationally demanding, so an approximation is considered: we remove the neuron with smallest initial residual. Notice that this approximation could lead to the same result of the original one.
Suppose (for simplicity) we have a trained network with one hidden layer and suppose that unit h is to be removed.

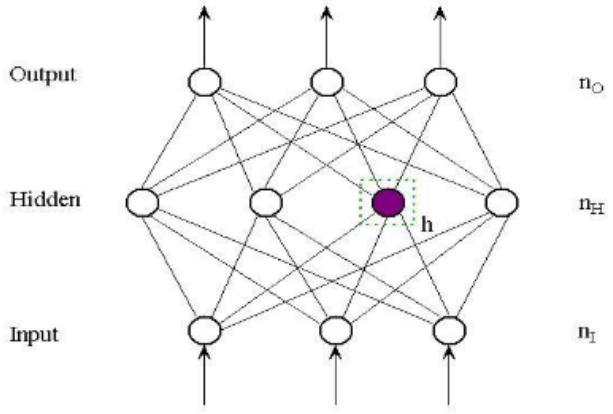


Figure 40: Example of the pruning approach.

As a consequence of this action we have to **remove** the **incoming/outgoing connections** and **update** the weights of the **other connections** such that the output remains the same. We focus on the input, since if the input remains the same also the output remains unchanged.
But this is equivalent to solving the following system of equations:

$$\underbrace{\sum_{j=1}^{n_h} w_{ij} y_j^{(\mu)}}_{\text{Before removing } h} = \underbrace{\sum_{\substack{j=1 \\ j \neq h}}^{n_h} (w_{ij} + \delta_{ij})\, y_j^{(\mu)}}_{\text{After removing } h} \qquad i = 1, \dots n_0,\ \mu = 1, \dots, P$$

, where:

- The left-hand size represents the input of the output layer **before** removing the unit h;

- The right-hand size represents the input of the output layer **after** removing the unit h;
- $i = 1 \dots n_0$ represent each of the output units (n_0 is the total number of output units);
- $\mu = 1 \dots P$ represent the examples in the training set;

However, we can rewrite the previous system as:

$$\begin{aligned}\sum_{j=1}^{n_h} w_{ij} y_j^{(\mu)} &= \sum_{\substack{j=1 \\ j \neq h}}^{n_h} (w_{ij} + \delta_{ij})\, y_j^{(\mu)} \\ &= \sum_{\substack{j=1 \\ j \neq h}}^{n_h} w_{ij} y_j^{(\mu)} + \sum_{\substack{j=1 \\ j \neq h}}^{n_h} \delta_{ij} y_j^{(\mu)} \\ &= \sum_{j=1}^{n_h} w_{ij} y_j^{(\mu)} - w_{ih} y_h^{(\mu)} + \sum_{\substack{j=1 \\ j \neq h}}^{n_h} \delta_{ij} y_j^{(\mu)}\end{aligned}$$

, from which we derive that

$$\sum_{j \neq h} \delta_{ij} y_j^{(\mu)} = w_{ih} y_h^{(\mu)} \qquad i = 1, \dots n_0,\ \mu = 1, \dots, P$$

We can observe that $\sum_{j \neq h} \delta_{ij} y_j^{(\mu)}$ is a linear system of equations and instead $w_{ih} y_h^{(\mu)}$ represents b, and they both depend on h. Clearly, our goal is to find the δ_{ij} s.t. the equality holds.
In a more compact notation, it is possible to write the previous linear system as:

$$Y_h \delta = b_h$$

, where:

- $Y_h \in \mathbb{R}^{P n_o \times n_o (n_h - 1)}$ is a matrix that represents the weights of the nodes in the output layer fed by the node h;
- δ represents the feature vectors;
- $b_h = w_{ih} y_h^{(\mu)}$ represents the contribution of the network.

However, the solution of this system does not always exist. For this reason, the **least squares solution** of the system is:

$$\min_{\delta} ||Y_h \delta - b_h||$$

In this sense, (informally) we choose the δ s.t. the difference of the network before and after removing h is minimum. Notice that we're now solving an easier problem, since it is a convex one.
A possible algorithm for solving the problem is the **residual-reducing algorithm**. This

method starts with an initial solution δ_0 and produces some sequences of points $\{\delta_k\}$ so that the residuals are computed as:

$$\begin{aligned} r_0 &= ||Y\delta_0 - b|| \\ r_1 &= ||Y\delta_1 - b|| \\ &\vdots \\ r_k &= ||Y\delta_k - b|| \qquad \text{where } r_k \leq r_{k-1} \end{aligned}$$

Usually, the starting point is in $\delta_0 = 0$, for which $r_0 = ||b||$.
In this sense, instead of solving the whole system, as we introduced before the algorithm considers only the initial contribution $||b||$ for each neuron, and then uses it as an estimate of the true contribution. Once the neuron with the smallest contribution is detected, the real linear system is computed and weights on the net are updated. All in all, this is an heuristic procedure that hopes that the initial contribution $||b||$ won't be too far from the real one.

Example $b = w_{ih}y_n$ and $A = \sum \delta_{ij} y_j^{(\mu)}$

$$\begin{aligned} h1 &: ||A_1x_1 - b_1|| = 0.1 \\ h2 &: ||A_2x_1 - b_2|| = 1.6 \\ h3 &: ||A_3x_1 - b_3|| = 7.6 \end{aligned}$$

From this example we can see that h_1 should be removed since the contribution of that neuron is the smallest one and it is very small. We can see that $||A_hx_h - b_h||$ measures how different the network is after removing neuron h. The problem of this procedure is that we have to solve many linear systems, one for each neuron.

An iterative pruning algorithm (Pelillo). The pruning algorithm follows these steps:

1. Start with an over-sized trained network;
2. Repeat

   2.1 Find the hidden unit h for which $||b||$ is minimum;
   2.2 Solve the corresponding system;
   2.3 Remove unit h;

   Until Perf(pruned) − Perf(original) $< \varepsilon$, where Perf() measures the quality of the network.
3. Reject the last reduced network.

Feature selection using pruning algorithms. The problem of **feature selection** is defined as follows: given a set of features, we want to **select** a **subset** of them. We can address this problem using network pruning.
In general, the classifiers are very sensitive to the features that are used, so it is quite important to remove irrelevant and redundant information, in order to both reduce the overfitting problem and to improve the generalization of the model. The idea for solving this problem, as we said before, is to apply a **pruning algorithm** on the input layer.
An example of application of such technique on the MNIST dataset lead to the the following results: as we can see, on average the pixels around the border are less significant.

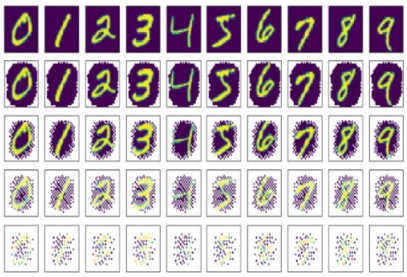


Figure 41: Example of feature selection on MNIST dataset

Optimal Brain Surgeon algorithm (OBS). This is another pruning algorithm, which focuses on the removal of a single connection, and it rescales all the weights in the network.
Consider a network which is trained to a local minimum in error E, i.e. the output of the back-propagation algorithm is $\min_{w \in \mathbb{R}^n} E(w)$. Then, we have that:

$$\delta E = (\frac{\partial E}{\partial w})^T \cdot \delta w + \frac{1}{2} \delta w^T \cdot H \cdot \delta w + O(||\delta w||^3)$$

, where:

- $\delta E = E(w) - E(w + \delta w)$;
- $(\frac{\partial E}{\partial w})^T \cdot \delta w \approx 0$, because we're choosing the minimum w, so the partial derivative of E is 0;
- $O(||\delta w||^3) \approx 0$;
- $H = \frac{\partial^2 E}{\partial w^2}$ represents the Hessian matrix, containing all the second derivative information.

In this sense, removing a single connection means setting one of the weights, which will be called w_q, to 0, so as to minimize the increase in error δE. Thus, eliminating w_q can be expressed as:

$$\delta w_q + w_q = 0$$

or

$$e_q^T \cdot \delta w + w_q = 0$$

, where e_q is the unit vector in the weight space corresponding to w_q, i.e. $e_q = [0..1..0]$, where the 1 is in position q. The final goal is then to solve the following constrained optimization problem:

$$\min_q \min_{\delta w} \frac{1}{2}\delta w^T \cdot H \cdot \delta w \qquad \text{such that } e_q^T \cdot \delta w + w_q = 0$$

We can transform this constrained problem into an unconstrained one using the **Lagrangian** (and by adding some more variables):

$$L = \frac{1}{2}\delta w^T \cdot H \cdot \delta w + \lambda(e_q^T \cdot \delta w + w_q)$$

, where λ is the (undetermine) Lagrange multiplier. The goal now is to find the stationary points of L, i.e. points in which all its partial derivatives vanish. Tanking the partial derivatives and setting them to zero we obtain:

$$\frac{\partial L}{\partial \delta w} = H\delta W + \lambda e_q = 0$$

and

$$\frac{\partial L}{\partial \lambda} = e_q^T \delta w + w_q = 0$$

, from which we obtain that:

$$\delta w = -\frac{w_q}{(H^{-1})_{qq}} H^{-1} \cdot e_q$$

and

$$L = \frac{1}{2}\frac{w_q^2}{(H^{-1})_{qq}}$$

, where H^{-1} represents the inverse of the Hessian matrix, which is very computational demanding.
Finally, the algorithm proceeds as follows:

1. Train a reasonably large network to minimum error;
2. Compute H^{-1};
3. Find the q that gives the smallest $L = \frac{1}{2}\frac{w_q^2}{(H^{-1})_{qq}}$. If this candidate error increase is much smaller than E, then the q-th weight should be deleted, and we proceed to step 4., otherwise, go to step 5.;
4. Use the q from step 3. to update all the weights $\delta w = -\frac{w_q}{(H^{-1})_{qq}} H^{-1} \cdot e_q$, and go to step 2.;
5. No more weights can be deleted without large increases in E, so at this point we can re-train the network.

There are some differences between the two pruning algorithms we considered:

- In OBS we update all the weights;

- In OBS we assume that $O(||\delta w||^3) \approx 0$ and $(\frac{\partial E}{\partial w})^T \cdot \delta w \approx 0$;
- In OBS we assume that the network converges to the minimum;
- OBS algorithm is less efficient than Pelillo's.

3 Deep Neural Networks

The **philosophy** on which **Deep Learning** relies is to provide a new method for selecting and extracting the features that are provided as input to a classifier, by exploiting the data of the training set.
More specifically, instead of selecting manually good features and then using them to feed a classifier, Deep Neural Networks learn from the data a **feature hierarchy** from the initial pixel image in order to obtain a **classifier**: each layer extracts features from the output of the previous layer, and finally the training phase involves all the layers, jointly.

Figure 42: Example of deep learning process.

As we can see, each layer of the pipeline learns to extract features from the image/video/pixels (in general, from the data we have), and the **deeper** the layer is, the **more abstract** the extracted features are.

3.1 Shallow vs Deep Networks

Shallow architectures are **inefficient** at **representing deep functions**, since they are characterized by a limited number of hidden layers. A shallow network with a large single hidden layer can fit any function (i.e. it is a universal approximator), but on the other hand this increases significantly the number of parameters.
A **deep network** can **fit** functions **better** with less parameters than a shallow network, increasing the number of hidden layers but decreasing the number of required parameters. Deep networks try to simulate the brain's behavior, in which the electric signals propagate across different layers.

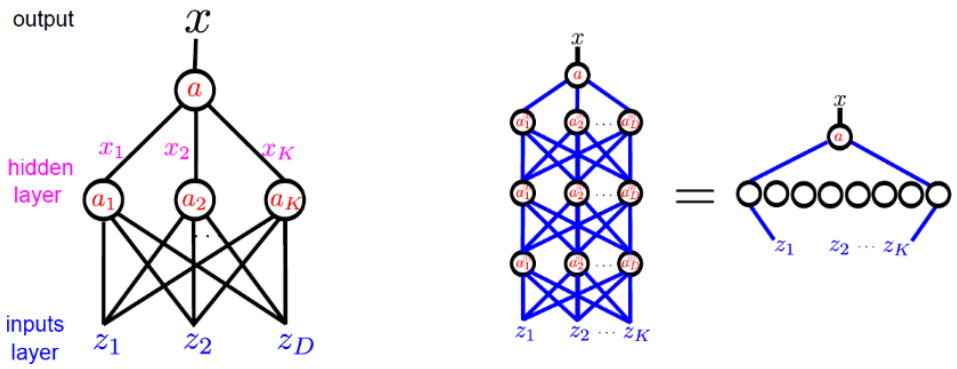


Figure 43: Shallow vs Deep Networks.

Another important aspect to notice is the **improvement** in **performance** with the presence of more data.

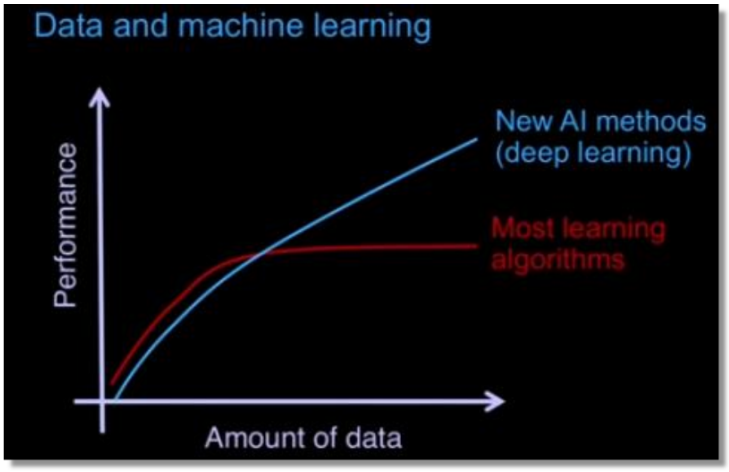


Figure 44: Performances improves with more data.

As we can see, in the case of basic ML algorithm after a certain amount of data the performance does not increase anymore: in SVM's, for example, this phenomenon happens since the decision boundary obtained by the model only depends on the support vectors, so increasing the size of the training set does not change the accuracy.
The **usage** of deep networks is not a **recent** idea (indeed, these networks are fairly old), but it has been made possible only nowadays thanks to the fact that we have **more data** and **more computing power**. In particular, the advances in the field of **GPUs** have made using deep networks way more feasible than before.

Image classification. The **image classification** problem consists in predicting a single label (or a probability distribution over all the possible labels to indicate our confidence, as per the following example) for a given image. Images are 3-dimensional arrays of integers from 0 to 255 of size Width x Height x 3. The 3 represents the three color channels Red, Green and Blue.

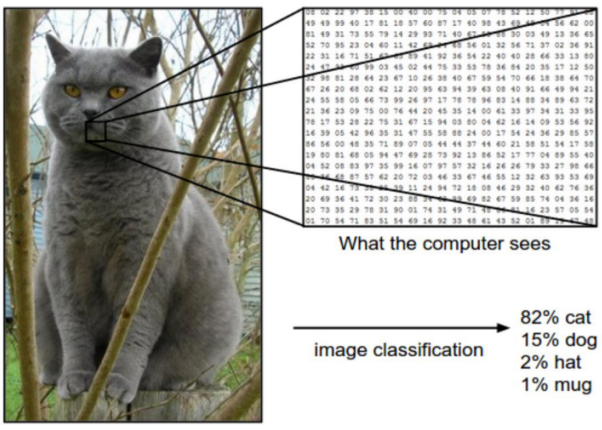


Figure 45: Image classification example.

In image classification, the most important **challenges** are the following ones:

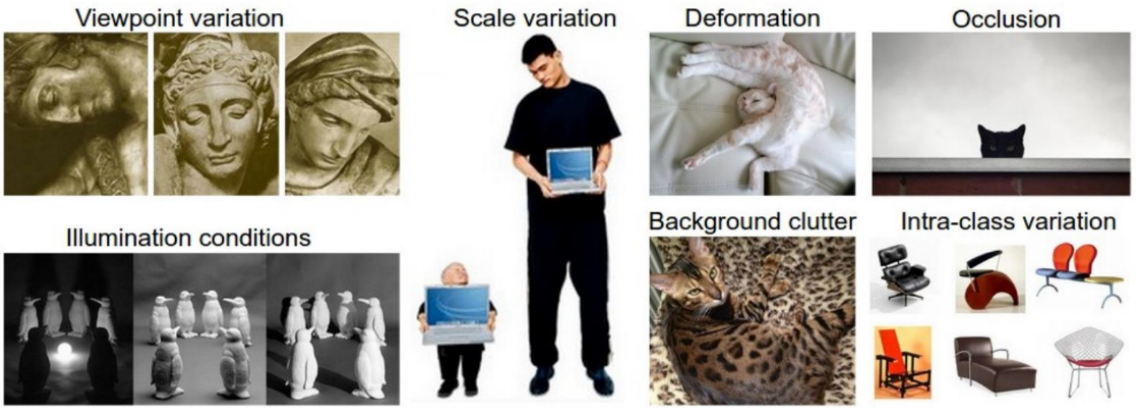


Figure 46: Challenges in image classification.

In order to face these challenges, what we need is a data-driven approach in which we have thousands of categories and hundreds of thousand of images for each category.
As it is possible to understand, there's a difference between **traditional approaches** and **deep learning**. Indeed, in the first one we extract **meaningful features** from images through a **manual** process, while in the second case everything happens **automatically** thanks to a sequence of **layers** in which the final ones are useful for classification.

Inspiration from biology. As we introduced before, functioning of Deep Neural Networks is highly inspired from biology, and in particular the architecture resembles the one in the visual cortex of the brain. Indeed, biological vision is hierarchically organized, and the basic component of this hierarchy is the **retina**. The cells of the retina are arrayed in discrete layers, with the **photoreceptors** at the top of them that are divided into:

- **Rods**, which are sensitive to the intensity of the light and to movements;
- **Cones**, which are sensitive to colors.

We define **receptive field** the region of the visual field in which light stimuli evoke responses of a given neuron. In terms of Deep Networks, this makes a distinction between **fully connected Neural Networks**, in which each neuron is connected to all the neurons of the previous layer, and **sparsely connected Neural Networks**, in which each neuron is characterized by a corresponding **receptive field**, i.e. it is connected only to a subset of neurons of the previous layer.
The **take-home** message of this digression is that the **visual system** is a **hierarchy** of **features detectors**.

The Neocognitron (1980). The first example example of self-trained network for feature learning was the **Neocognitron**, introduced in 1980.

3.2 Convolution

The **convolution** is a mathematical operation that takes as input an **image**, i.e. an array of pixels, and a 3x3 array of numbers (called **convolution filter**), and applies the

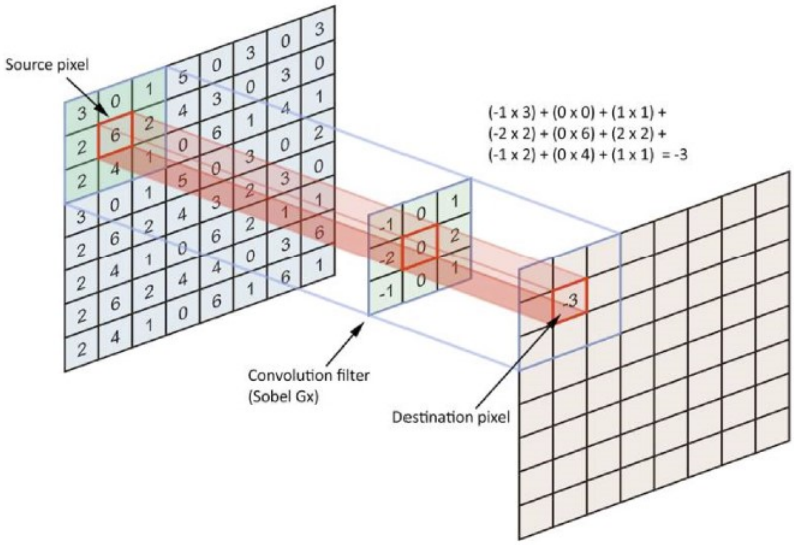


Figure 47: Example of convolution

3x3 array in a certain portion of the image, and computes the summation of the products between the pixels of the image and the one in the filter.
Notice that there exist many type of filters, which differ for their values.

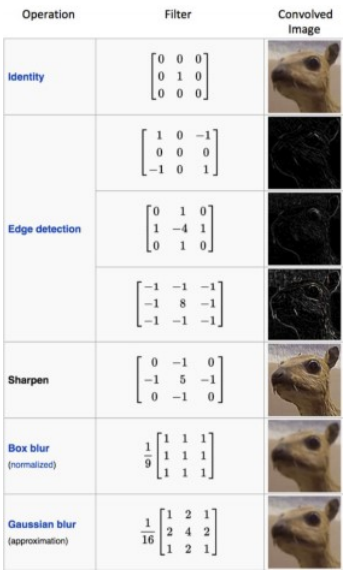


Figure 48: Types of filters

3.2.1 Stride and Padding

The **stride** quantity denotes how many steps we're moving in each step of convolution, and the default value is 1. In order to maintain the dimension of the output the same as the one of the input, we use **padding**, which is the process of adding 0s to the input matrix symmetrically.

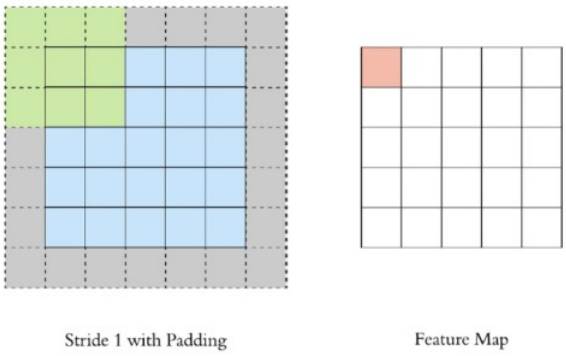


Figure 49: Stride = 1 with padding = 1

In this sense, **stride** and **padding** can be used to **adjust** the **dimensionality** of the **data** effectively.

3.2.2 Multiple channels

In the case in which the convolution is applied to an image with multiple channels, we can use a different filter for each channel, and then combine the results into a single output matrix.

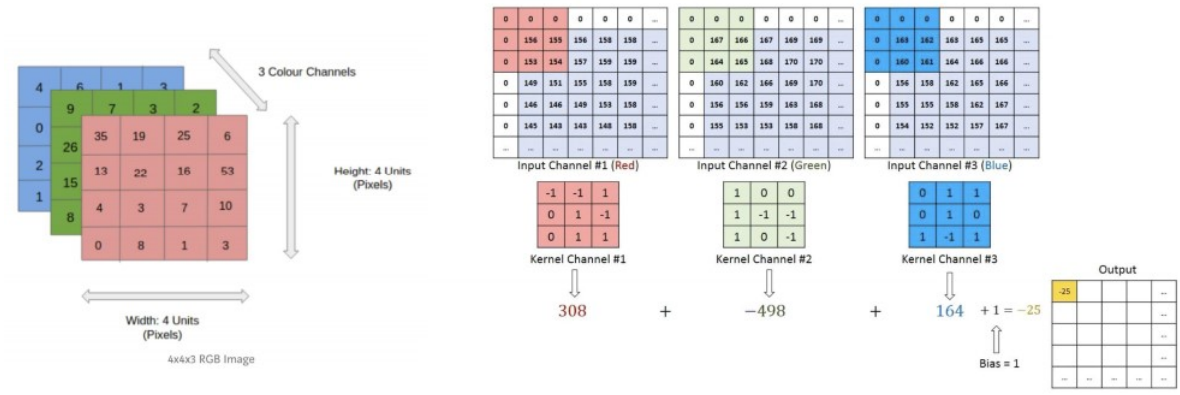


Figure 50: Example of convolution of an image with multiple channels

3.2.3 Gaussian filter

A very famous filter used in convolution is the Gaussian filter, which basically computes a weighted average of the pixels of the image, where the weights are proportional to the distance with the central pixel.
The result of applying a Gaussian filter to an image is to obtain a **blurred version** of the image: the larger the filter, the more blurred the output image is.

7×7 Gaussian mask

1	1	2	2	2	1	1
1	2	2	4	2	2	1
2	2	4	8	4	2	2
2	4	8	16	8	4	2
2	2	4	8	4	2	2
1	2	2	4	2	2	1
1	1	2	2	2	1	1

Figure 51: Example of 7x7 Gaussian filter

3.2.4 Convolution for edge detection and other problems

An important application of the convolution operation is **edge detection**, i.e. the problem of determining the contour of an object.

Figure 52: Edge detection problem

Some of the most famous filters that are used in edge detection are *Roberts operator*, *Sobel operator* and *Prewitt operator*. Other filters are used for different tasks, e.g. *HoG* is used for the problem of *pedestrian recognition* etc..

3.3 Convolutional Neural Networks (CNNs)

3.3.1 Fully-connected and sparsely-connected networks

A neural network can be defined as a **fully-connected network** or a **locally-connected network**. In a fully-connected network, each neuron of each layer is connected to every neuron in the previous one, and each connection has its own weight. In this sense, the number of parameters is huge.
Conversely, in a locally-connected layer, each neuron is only connected to a **few nearby neurons** in the previous layer, and the same set of weights (and local connection layout) is used for each neuron. The typical use case for locally-connected layers is for image data where, as required, the **features** are **local** (e.g. a "nose" consists of a set of nearby pixels,

which are not spread across the whole image), or in general in applications where the local connections capture local dependencies. The smaller number of connections and weights makes local-connected layers **relatively cheap** in terms of memory and computing power needed.

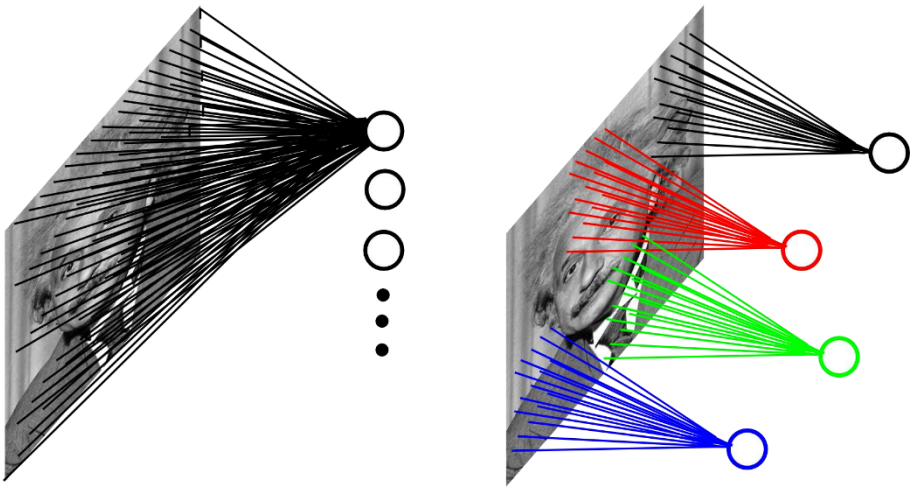


Figure 53: Fully vs local-connected networks.

3.3.2 Weight sharing

Before providing the definition of **CNN**, we now define the concept of **weight sharing**. This concept is based on the following reasonable assumption: if one feature is useful to compute at some spatial position (x_1, y_1), then it should be useful to compute at a different position (x_2, y_2). In this way we can dramatically reduce the number of parameters.

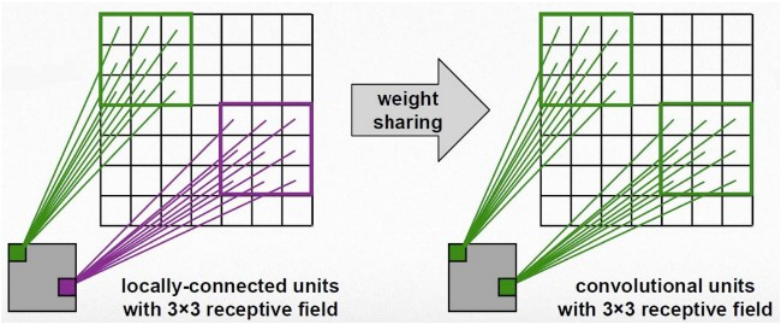


Figure 54: Edge detection problem

As we can see, in the first case we have neurons detecting different features (i.e. different weights in the receptive fields), while in the second case the two neurons have the same weights in the corresponding receptive fields, so they're detecting the presence of the same feature in different portions of the image. Note that while in the traditional convolution operation the weights are applied directly to the pixels of the image, in this case they become the weights of the connections of the receptive fields.

3.3.3 Definition of CNN

A **Convolutional Neural Network (CNN)** is a **multi-layer feed-forward neural network** characterized by **local connectivity** (i.e. neurons with the correspondent receptive field) and **weights** that are **shared** across spatial positions. Normally, **several filters** are **packed** together and **learned automatically** during training.

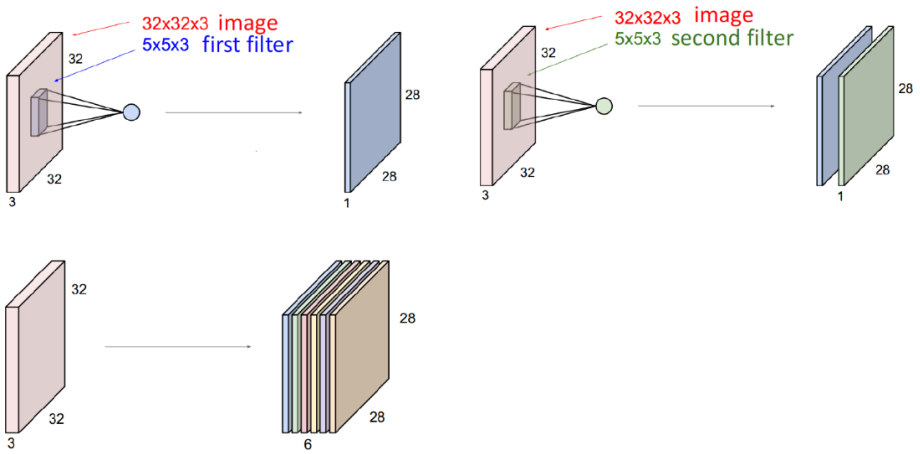


Figure 55: Using Several Trainable Filters.

Pooling. **Pooling** is a way to **simplify** the network architecture, by downsampling the **number of neurons** resulting from the filtering operations. An example of pooling technique is the **max pooling**, in which the **image** is **partitioned** in small squares and for each square the pixel with **maximum value** is taken.

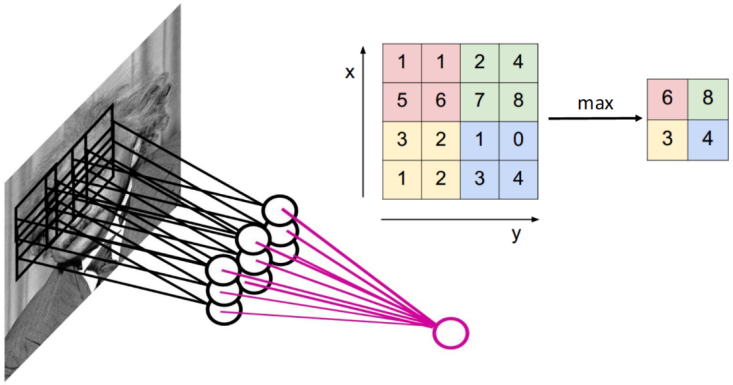


Figure 56: Max Pooling.

As we can see in the next image, a **Deep Convolutional Neural Network** is a **combination** of **feature extraction** and **classification processes**:

- The **input** of the network is represented by an **image**;
- The **input layer** is followed by some **locally-connected layers** that **extract features** from the image. As we can see, this feature extraction phase is mainly composed of *convolution* and *subsampling* operations;

- Finally, a **fully-connected layer** provides the **classification** of the input image.

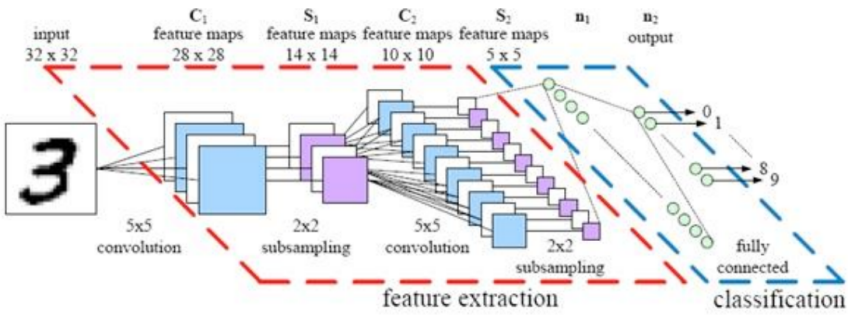
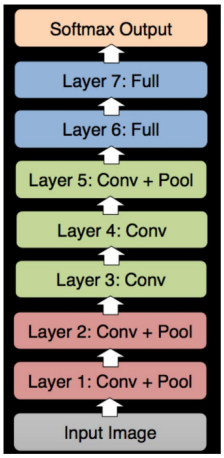


Figure 57: Combining Feature Extraction and Classification.

3.4 AlexNet (2012)

The first example of Deep Convolutional Neural Network we examine is the **AlexNet**. This architecture was developed in 2012 for solving the problem of **image classification**, and it was trained using the *ImageNet* dataset, comprising 1,000 categories, 1.2M training images and 150k test images. The results were astonishing, since the error was reduced by 22% in only 3 years.
More specifically, this architecture is not so different from the one of the *Neocognitron*, or from the one proposed by LeCun in 1998, but this was much larger. AlexNet is composed of **8 layers** as per the following schema:



The first thing we can notice are that:

- The **input** is given by an **image**;
- The **first 5 layers** are used for **extracting features** for the classification phase: as we can see, we have a combinations of *convolution* and *pooling* operations;
- **Layer 6 and 7** provide the **classification** of the input: as we can see, in this case we exploit fully connected layers;
- The **output layer** is represented by a **Softmax** function, which provides a probability distribution of the classes of the input image.

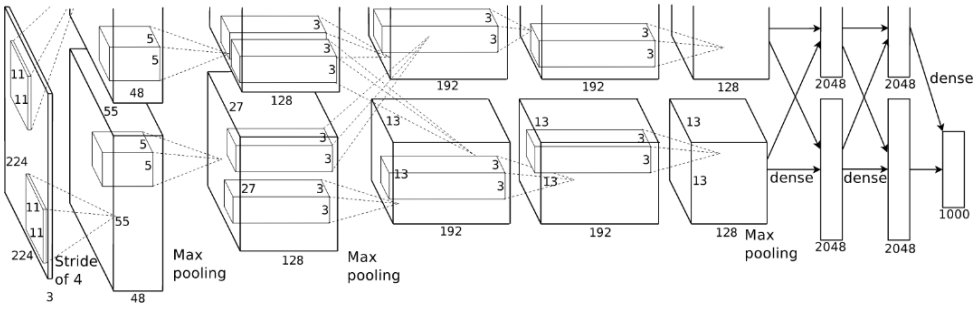


Figure 58: AlexNet architecture.

Diving deeper into these layers' characteristics, we can notice that:

- **1st layer:** we have 96 kernels of size $(11 \times 11 \times 3)$, which are applied to the input image: the results is then convolved (stride $= 4$) and pooled. Notice that the output of the first layer has not a width of 96, and this is because it was splitted into two different outputs, each with width equal to 48. By looking at the top-9 patches for one filter we can observe that the neurons are very sensitive for colors and geometric forms, so they're very simple;
- **2nd layer:** we have 256 kernels of size $(5 \times 5 \times 48)$, which are then normalized and pooled. Notice the 256 kernels are given by two blocks of 128 kernels. Here the neurons distinguish more abstract features, and this process continues as we go deeper in the Deep Network;
- **3rd layer:** we have 384 kernels of size $(3 \times 3 \times 256)$;
- **4th layer:** we have 384 kernels of size $(3 \times 3 \times 192)$;
- **5th layer:** we have 256 kernels of size $(3 \times 3 \times 192)$;
- **6th layer:** we have a fully connected layer with 4096 neurons;
- **7th layer:** we have a fully connected layer with 4096 neurons;
- **8th layer:** we have a 1000-way **SoftMax** output layer, i.e. 1 output for each of the possible classes, since it provides a probability distribution.

While training the network, two independent GPUs run in parallel, in order to speedup the training process. This is the reason why the output of the first layer was splitted.
The last *SoftMax* layer gives as output:

$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

, where z_i represents the output of the network and it is defined as:

$$z_i = w_i^T \cdot x$$

Since the output of the SoftMax is a probability distribution over all the possible classes of the training set, we can compute the **Cross-entropy loss** between the actual output and the desired one, in order to measure the quality of the model.

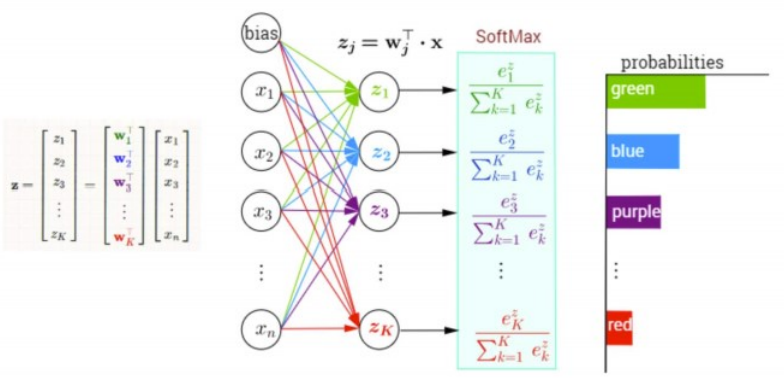


Figure 59: Output of the SoftMax function

3.5 ReLU

ReLU is an acronym that stands for *Rectified Linear Unit*. ReLUs are used to solve the **problem** that **sigmoid activation** takes only values in $(0, 1)$. While propagating the gradient back to the initial layers, it tends to get closer and closer to 0 (this phenomenon is called *vanishing gradient*, and it intensifies when the number of layers is high). From a practical perspective, this slows down the training procedure of the initial layers of the network. Indeed, with sigmoid the gradient will be close to 0 and the algorithm won't learn. In order to speed up the learning phase, ReLU is used.

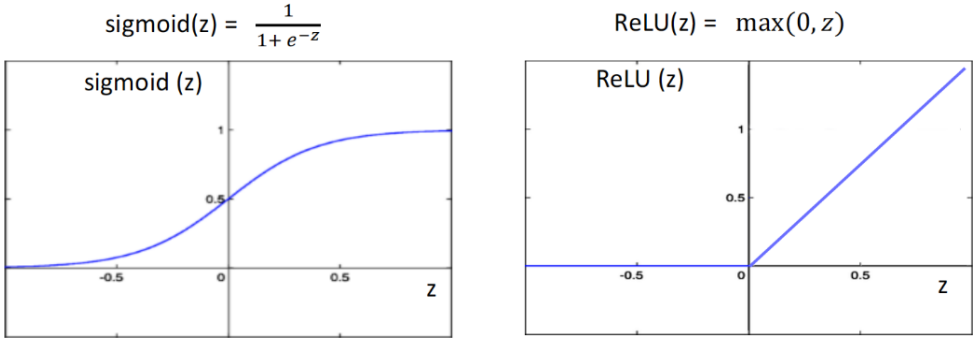


Figure 60: Comparison between sigmoid and ReLU functions.

It is also possible to notice that ReLU reaches the **same results as sigmoid**, but **much faster**. In the following image, the solid line represents the convergence of a 4 layer CNN with ReLUs, while the dashed line represents an equivalent network with *tanh* neurons. It can be noticed that the CNN with **ReLUs converges six times faster**.

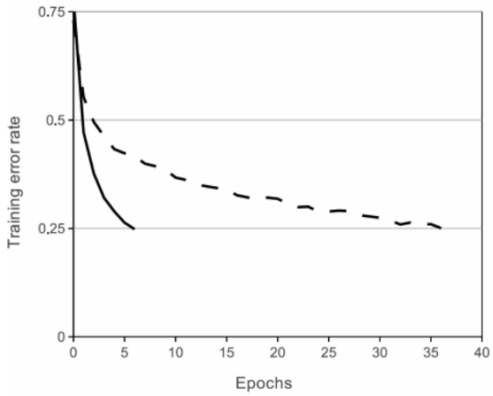


Figure 61: Convergence of ReLU neurons.

3.6 Mini-batch Stochastic Gradient Descent

We recall that in the back-propagation algorithm we can have either the **on-line** implementation or the **off-line** one: in the first one, the weights are updated at any presentation of an example of the training set, while in the second one the gradient is computed exactly. We indicated the **stochastic gradient descent** as a possible compromise between the two approaches, now we introduce the **mini-batch stochastic gradient descent**, which is defined as follows:

1. Sample a batch of data (the dimension is an hyperparameter to choose), i.e. a subset of the training set;
2. Perform the forward pass and compute the loss;
3. Perform the backward pass to compute the gradients;
4. Update the weights using the gradient (by minimizing the loss).

Notice that, like in stochastic gradient descent, the use of random samples helps in finding global minima.

3.7 Data augmentation

The easiest and most common method to **reduce overfitting** on image data is to **artificially enlarge** the **dataset** using **label-preserving transformations**. This technique is used to make the dataset more robust. **AlexNet** uses two forms of **data augmentation**:

- The first form consists in **generating image translations** and **horizontal reflections** (translations, scaling or rotations of the images);
- The second form consists in **altering the intensities** of the **RGB channels** of the **training images** (introduce random noise inside the images).

Figure 62: Data augmentation: translation, reflection, scaling, rotation, RGB noise.

3.8 Dropout

This technique is exploited in order to have a network with **more generalization power**, and it consists on **randomly dropping out some neurons** during the back-propagation algorithm, by setting to 0 the output of that neuron with probability 0.5. The neurons which are *"dropped out"* in this way do not contribute to the forward pass and do not participate in back-propagation. Every time an input is presented, the neural network samples a different architecture, but all these architectures share weights.
Dropout reduces complex co-adaptations of neurons, and it forces the network to find different "roads" in order to produce the outputs, resulting in a more robust network.

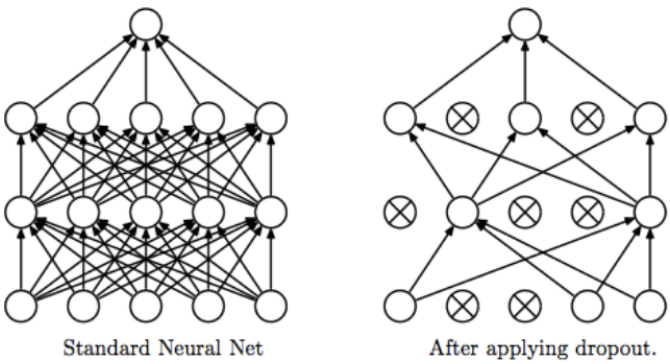


Figure 63: Dropout results

3.9 Feature analysis

A well trained Deep Convolutional Neural Network is an excellent **feature extractor**: in particular, the idea is to chop the network at a desired layer and use the output as a feature representation to train a SVM on some other dataset. Here we show some results.

	Cal-101 (30/class)	Cal-256 (60/class)
SVM (1)	44.8 ± 0.7	24.6 ± 0.4
SVM (2)	66.2 ± 0.5	39.6 ± 0.3
SVM (3)	72.3 ± 0.4	46.0 ± 0.3
SVM (4)	76.6 ± 0.4	51.3 ± 0.1
SVM (5)	$\mathbf{86.2 \pm 0.8}$	65.6 ± 0.3
SVM (7)	$\mathbf{85.5 \pm 0.4}$	$\mathbf{71.7 \pm 0.2}$
Softmax (5)	82.9 ± 0.4	65.7 ± 0.5
Softmax (7)	$\mathbf{85.4 \pm 0.4}$	$\mathbf{72.6 \pm 0.1}$

Figure 64: Results using CNN's as feature extractors

The parenthesis indicates the layer at which the output is taken: as we can see, the accuracy of the classifier increases as we choose features from more layers, underlying the property of CNN's of learning more and more accurate features as the number of layers grow.

3.10 CNN's in computer vision tasks

After 2012, many CV tasks were addressed using CNN's, for example:

- **Semantic segmentation**: in this case the input is an image, and each pixel is classified with a label;
- **Classification and localization**: in this case we assume that the input image contains a single prominent object, and the output consists of single label of the object (standard classification) together with the position of the object;
- **Object detection**: in this case we detect all the objects in an image, where the number is not known in advance;
- **Instance segmentation**: this problem represents a variation of the image segmentation problem, and it consists of labeling each pixel differentiating between objects having the same label.

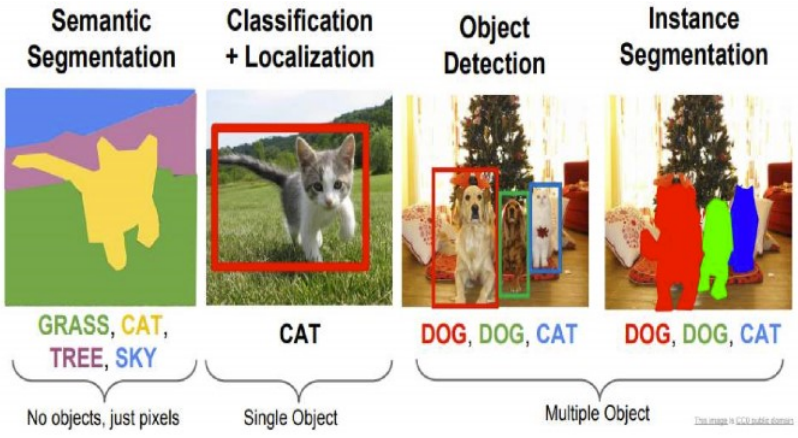


Figure 65: Computer vision problems

In general, the adoption of CNN's for solving such problems is having a huge impact in the results.

3.11 Recurrent Neural Networks

If we consider the problem of **image captioning**, i.e. of providing a textual description of the content of an image, it is clear that the usage of a **feed-forward neural network** does not help in this case, since the output depends on the input. For this reason, we have to exploit **recurrent neural networks**.

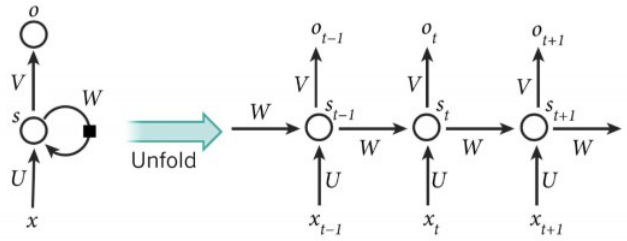


Figure 66: Recurrent neural network

The image shows the unfolding of the simplest recurrent NN, i.e. the one composed of a single hidden layer. Notice that this architecture is different from a feed-forward NN for two main reasons:

- We do not know in advance the number of layers;
- In a feed-forward NN we have different weights between the layers, while in this case we only have 1 batch of weights for each connection.

If we denote with W_{hx} the weights between the input vector x and the RNN, with W_{hh} the weights of the RNN and with W_{hy} the weights between the RNN and the output vector, then:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

and

$$y_t = W_{hy}h_t$$

, where h_t represents the output of the hidden layer at time t.
We can have many types of RNN, as shown in the following image.

- The **one-to-many** RNN is used for solving the **image captioning** problem;
- The **many-to-one** RNN is used for solving the **sentiment classification** problem (i.e., given a sequence of words, provide the sentiment);
- The **many-to-many** RNN is used for solving the **machine translation** problem (i.e., given a sequence of words, translating it into another sequence of words). The many-to-many is also used for **video classification**.

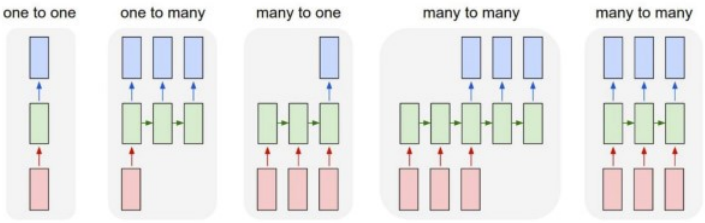


Figure 67: Recurrent neural network

3.11.1 Character-level Language Model

An example of application of RNN is the character-level language model, in which the goal is to predict the following letter.

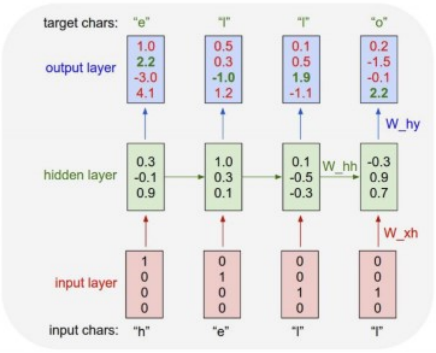


Figure 68: Character-level LM: training

As we can see, at test time the predictions at time t are used for the predictions at time $t + 1$.

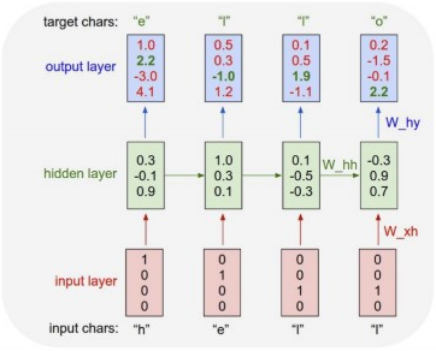


Figure 69: Character-level LM: test

However, one possible problems when dealing with RNN is the **vanishing gradient problem**, so a possible solution is to perform a **truncated back-propagation**: in this way, the error is backpropagated for a smaller number of steps.

3.11.2 Image captioning

We focus now on the image captioning problem: the idea here is to exploit a CNN (in this case AlexNet) to extract the features of an image, and then use these features for training a RNN which can solve the problem.

$$\text{Image} \rightarrow \text{CNN} \rightarrow \text{RNN}$$

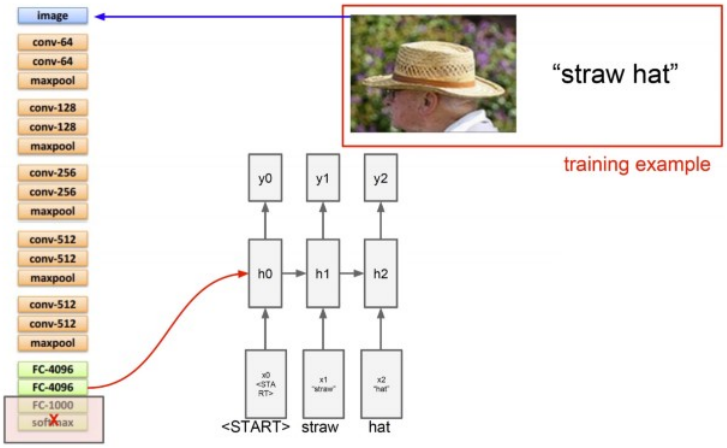


Figure 70: Image captioning: training

As we can see, the last two layers are deleted, since we do not care about the actual classification of the image, but only on the feature the CNN extracted. Finally, the features are provided as input to the RNN, which returns the caption of the image.

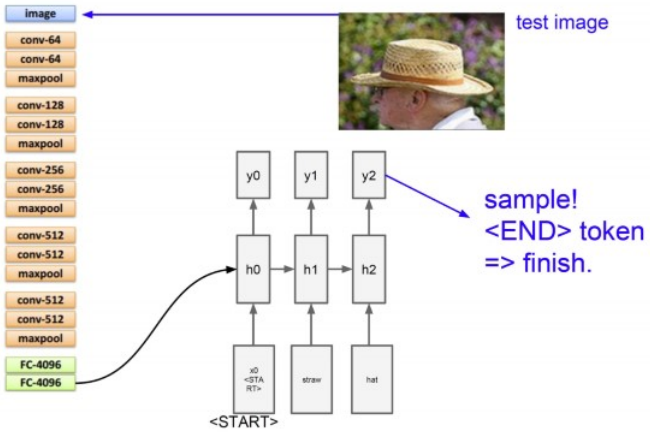


Figure 71: Image captioning: test

At test time, an image without caption is provided as input to the CNN, the features are extracted, and then the RNN is used to provide the caption of the image.

3.12 Some problems of CNN

It was showed in several papers how adding some noise bring the Deep Neural Network to misclassify the input image, and this inspired a very important branch of ML, called **adversarial ML**.

4 Statistical Learning Theory

STL mainly deals with **supervised learning** problems: given an input (feature) space $\mathcal{X}$ and an output (label) space $\mathcal{Y}$ (typically $\mathcal{Y} = \{+1, -1\}$), the goal is to estimate a functional relationship between the input and output space:

$$f : \mathcal{X} \to \mathcal{Y}$$

Usually, f is called **classifier**, so a **classification algorithm** is a procedure that takes the training data $((X_1, Y_1), .., (X_n, Y_n) \in \mathcal{X} \times \mathcal{Y})$ as input, and provides the classifier f as output.

4.1 Assumptions

Moreover, SLT makes the following **assumptions**:

1. There exists a joint probability distribution P among $\mathcal{X} \times \mathcal{Y}$, which is usually not known;
2. The training examples (X_i, Y_i) are sampled i.i.d. from P.

In particular,

- In STL, no assumptions are made on P, whereas in *statistical inference* the data are assumed to follow a certain distribution;
- The distribution of P is unknown at learning time;
- Non-deterministic labels due to label noise or overlapping classes;
- The distribution of P is fixed, both during training and testing.

4.2 Losses and risks

Clearly, we need to have some measure of how good a function f is when used as a classifier, so a **loss function** measures the "cost" of classifying instance $x \in \mathcal{X}$ as $y \in \mathcal{Y}$. In this sense, the simplest loss function is the **0-1 loss**, which is defined as:

$$l(X, Y, f(X)) = \begin{cases} 1 & \text{if } f(X) \neq Y \\ 0 & \text{otherwise} \end{cases}$$

We can define the **theoretical risk** of a function f the average loss over data points generated according to the underlying distribution P:

$$R(f) := \mathbb{E}(l(X, Y, f(X)))$$

The **best classifier** is the one with the smallest risk $R(f)$: among all possible classifiers, the best one is the *Bayes classifier*:

$$f_{\text{Bayes}} := \begin{cases} 1 & \text{if } P(Y = 1|X = x) \geq 0.5 \\ -1 & \text{otherwise} \end{cases}$$

Its idea is to classify the most frequent class. However, in practice it is impossible to directly compute the Bayes classifier, since the underlying distribution P is unknown to the learner, and estimating P from the data usually doesn't work.
Recall: Bayes' theorem says that:

$$P(h|e) = \frac{P(e|h)P(h)}{P(e)} = \frac{P(e|h)P(h)}{P(e|h)P(h) + P(e|\bar{h})P(\bar{h})}$$

, where:

- $P(h)$ represents the **prior probability** of hypothesis h;
- $P(h|e)$ represents the **posterior probability** of h after the evidence e;
- $P(e|h)$ represents the **likelihood** of evidence e on hypothesis h.

Returning to the classification problem, now the situation is that given:

- A set of training data $(X_1, Y_1), .., (X_n, Y_n) \in \mathcal{X} \times \mathcal{Y}$ drawn i.i.d. from an *unknown* distribution P;
- A loss function,

the goal is to determine a function $f : \mathcal{X} \to \mathcal{Y}$ which has a risk $R(f)$ which is as close as possible to the risk of the Bayes classifier. However, we notice that it is impossible to compute the risk of f without knowing P..

4.3 The nearest neighbor (NN) rule

A possible solution for this problem could be represented by the **nearest neighbor** approach, according to which the label of a data point x is given by the label of the nearest point to x. Notice that in this case, no assumptions about the probability distribution of the data is used, since the method only uses information from the training set.
But, how good is the NN rule? It was showed that:

$$R(f_{\text{Bayes}}) \leq R_\infty \leq 2R(f_{\text{Bayes}})$$

, where R_∞ denotes the expected error rate of NN when the sample size tends to infinity. Notice that we cannot say anything stronger about the bounds, since there are probability distributions for which the performance of the NN rule achieves either the upper or lower bound.
There exist some variations to the NN rule, for example the k**-NN** rule, which uses k nearest neighbors and takes the majority vote, or the k_n**-NN rule**, which does the same, but for k_n growing with n.
Theorem (Stone, 1977): if $n \to \infty$ and $k \to \infty$, such that $k/n \to 0$ (i.e. n grows faster than k), then for all probability distributions, $R(k_n - \text{NN}) \to R(f_{\text{Bayes}})$, i.e. the $k_n - \text{NN}$ rule is universally Bayes consistent.
However, all these NN rule have some **disadvantages**:

- Not having a learning phase (*lazy algorithm*), a huge amount of data must be kept in memory;
- They are very time demanding.

4.4 The kernel rule

The idea of **kernel rules** is that rather than fixing the number of neighbors, to classify a new point x we might consider fixing a distance h and taking a majority vote among the labels of all examples that fall within a distance h of x.

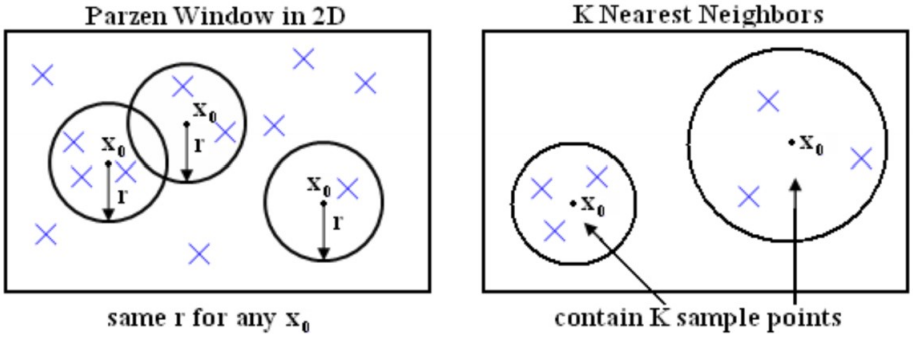


Figure 72: Difference between kernel rules and NN rule

Notice that in NN rule, the number of neighbors is fixed, while in kernel rule it depends on the value of h. The parameter h is usually called **smoothing factor** (or **bandwidth**). A **kernel** (a.k.a. *Parzen windows*) is defined as:

$$K(\bar{x}) = \begin{cases} 1 & \text{if } ||\bar{x}|| \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

and we define the vote counts as:

$$v_n^0(\bar{x}) = \sum_{i=1}^{n} I_{\{y_i=0\}} K(\frac{\bar{x} - \bar{x}_i}{h})$$

, i.e. the sum of all the elements with label 0 that are within $\bar{x}$ and $\bar{x}_i$, and

$$v_n^1(\bar{x}) = \sum_{i=1}^{n} I_{\{y_i=1\}} K(\frac{\bar{x} - \bar{x}_i}{h})$$

, i.e. the sum of all the elements with label 1 that are within $\bar{x}$ and $\bar{x}_i$.
Then, we assign x to the class 0 if and only if $v_n^0(x) \geq v_n^1(x)$.
A basic window kernel is represented in Picture 4.4, while Picture 4.4 shows other possible kernels that can be used.

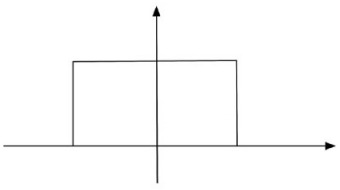
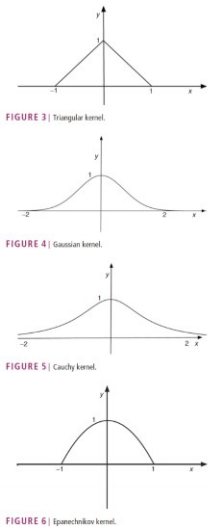


Figure 73: Basic window kernel



Triangular kernel: $K(\overline{x}) = (1 - \|\overline{x}\|)I_{\{\|\overline{x}\| \leq 1\}}$

Gaussian kernel: $K(\overline{x}) = e^{-\|\overline{x}\|^2}$

Cauchy kernel: $K(\overline{x}) = 1/(1 + \|\overline{x}\|^{d+1})$

Epanechnikov kernel: $K(\overline{x}) = (1 - \|\overline{x}\|^2)I_{\{\|\overline{x}\| \leq 1\}}$.

FIGURE 3 | Triangular kernel.

FIGURE 4 | Gaussian kernel.

FIGURE 5 | Cauchy kernel.

FIGURE 6 | Epanechnikov kernel.

Figure 74: Other possible kernels

4.5 Empirical Risk Minimization (ERM)

At the end of 1960's, the **Empirical Risk Minimization** theory was introduced, and it is based on the following **principle**: instead of looking for a function which minimizes the true risk $R(f)$, we try to find one which minimizes the **empirical risk**, i.e. the error the model makes on the training data, which is defined as:

$$R_{\text{emp}} = \frac{1}{n} \sum_{i=1}^{n} l(X_i, Y_i, f(X_i))$$

Given a training data $(X_1, Y_1), .., (X_n, Y_n) \in \mathcal{X} \times \mathcal{Y}$, a function space $\mathcal{F}$ and a loss function, we define the classifier f_n as:

$$f_n := \arg\min_{f \in \mathcal{F}} R_{\text{emp}}(f)$$

, i.e. we choose from $\mathcal{F}$ the function that minimizes the empirical risk. This approach is called the **empirical risk minimization (ERM)** induction principle, and is motivated from the law of large numbers.
However, we have the issue of how to choose the function space $\mathcal{F}$: a fundamental result in SLT is that the set of rules in $\mathcal{F}$ cannot be too rich, where the richness of $\mathcal{F}$ is measured by its **VC dimension**.

4.6 Estimation vs approximation

Before proceeding, let us introduce a few concepts:

- **Bayes error**: error of the best possible predictor (i.e. the Bayes predictor);
- **Approximation error**: error related to the type of model we are assuming (i.e. related to $\mathcal{F}$). The model may not reflect the nature of the underlying probability distribution. In other words, the approximation error is the minimum generalization error achievable by a predictor in $\mathcal{F}$, and it depends only of $\mathcal{F}$;

- **Estimation error**: error of a predictor belonging to family $\mathcal{F}$. Each predictor in the family will bring an additional error to the approximation error. In other words, the estimation error is the difference between the error of the considered predictor and the error of the predictor, belonging to $\mathcal{F}$, which minimizes the training error (the "best" predictor). The quality of this estimate depends on both the (size of) the training set and on the complexity of the hypothesis class $\mathcal{F}$.

Ideally we want to make $R(f_n) - R(f_{Bayes})$ as small as possible, as $n \to \infty$. Denoting by $f_{\mathcal{F}}$ the best classifier in $\mathcal{F}$, the difference can be decomposed as:

$$R\left(f_n\right) - R\left(f_{Bayes}\right) = \underbrace{\left(R\left(f_n\right) - R\left(f_{\mathcal{F}}\right)\right)}_{\text{estimation error}} + \underbrace{\left(R\left(f_{\mathcal{F}}\right) - R\left(f_{\text{Bayes}}\right)\right)}_{\text{approximation error}}$$

, where

- $R(f_n)$ is the risk of the considered classifier;
- $R(f_{\mathcal{F}})$ is the risk of the best classifier f on the family $\mathcal{F}$;
- $R\left(f_{\text{Bayes}}\right)$ is the risk of the best classifier overall (Bayes).

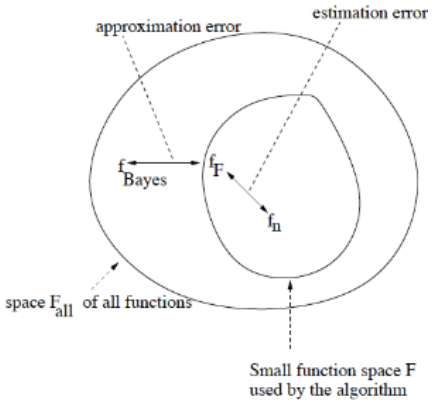


Figure 75: Estimation vs approximation.

According to the complexity of $\mathcal{F}$ we can have:

- **small complexity** of $\mathcal{F}$: small estimation error (small **variance**), large approximation error (large **bias**), resulting in *underfitting*;
- **large complexity** of $\mathcal{F}$: large estimation error (large **variance**), small approximation error (small **bias**), resulting in *overfitting*.

The best overall risk is achieved for "moderate" complexity.

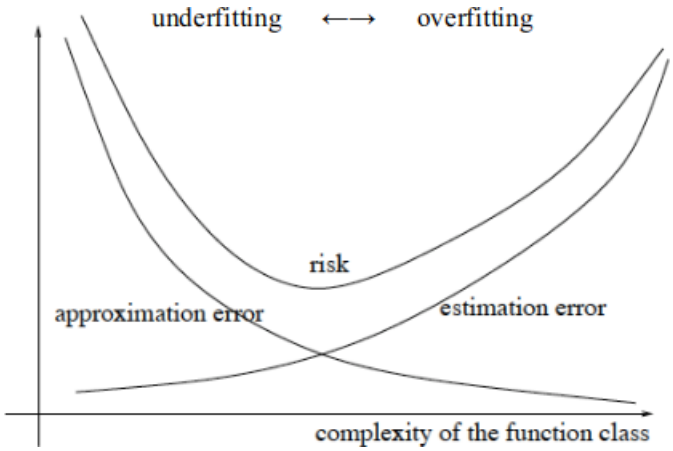


Figure 76: Underfitting vs Overfitting graph

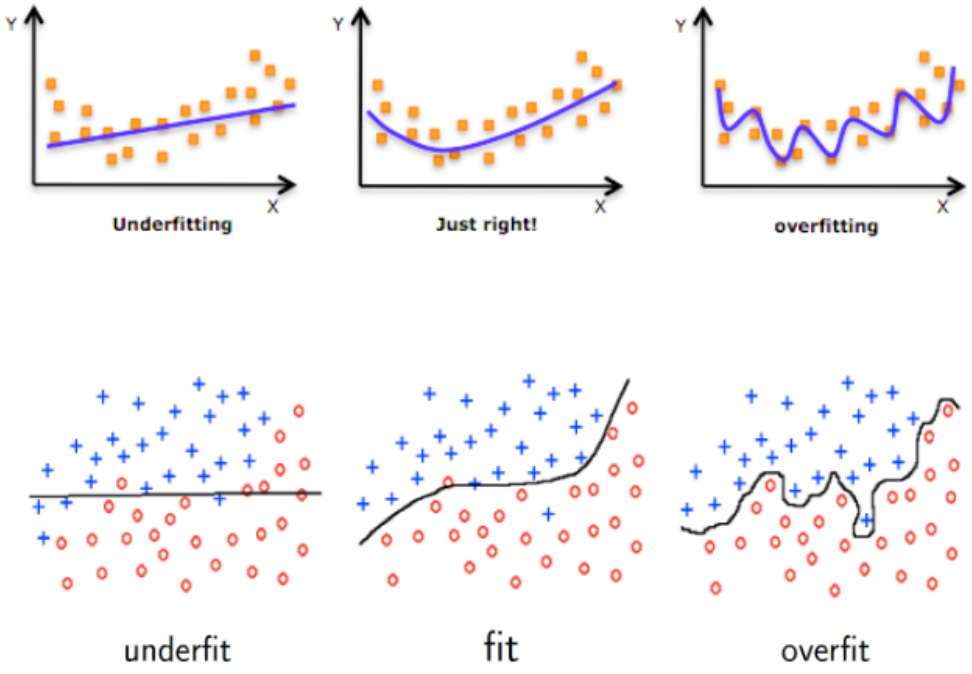
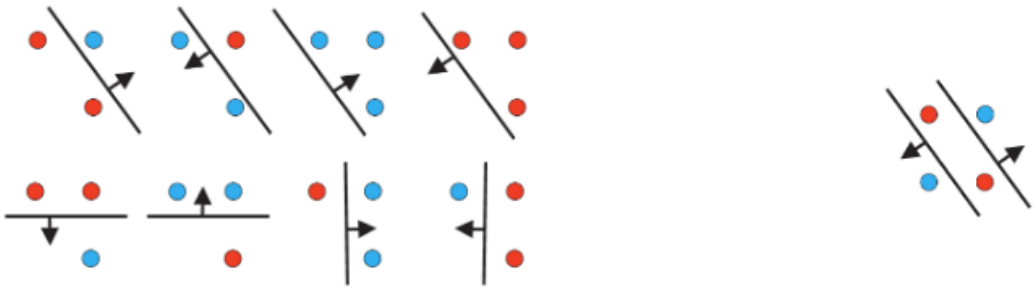


Figure 77: Model selection.

Shattering. A set of n instances $x_1, \ldots, x_n$ from the input space $\mathcal{X}$ is said to be *shattered* by a function class $\mathcal{F}$ if all the 2^n labelings of them can be generated using functions from $\mathcal{F}$.
For instance, with $\mathcal{F}$ as a linear decision functions (straight lines) in the plane, we can have:

(a) Any set of 3 non-collinear points shatters $\mathcal{F}$

(b) No set of 4 points can shatter $\mathcal{F}$



The Vapnik–Chervonenkis dimension. The **VC dimension** of a function class $\mathcal{F}$, denoted $VC(\mathcal{F})$, is the largest integer h such that there exists a sample of size h which is shattered by $\mathcal{F}$. It is a measure of complexity of a function class.
If arbitrarily large samples can be shattered, then $VC(\mathcal{F}) = \infty$.
For example:

- $\mathcal{F}$ = linear decision functions in $\mathbb{R}^2 \rightarrow VC(\mathcal{F}) = 3$
- $\mathcal{F}$ = linear decision functions (hyperplanes) in $\mathbb{R}^n \rightarrow VC(\mathcal{F}) = n + 1$
- $\mathcal{F}$ = multi-layer perceptrons with W weights $\rightarrow VC(\mathcal{F}) = O(W \ \log(W))$
- $\mathcal{F}$ = nearest neighbor classifiers $\rightarrow VC(\mathcal{F}) = \infty$

The VC dimension of a classifier depends on the dimension of the space the data points belong to. For instance, if we consider our space to be $\mathbb{R}^2$, then the VC dimension is 3. As a matter of fact, $\mathbb{R}^2$ can always shatter any three general position points ("general position" means they do not coincidentally lie on the same line). For instance, consider three points $(0,0)$, $(0,1)$, $(1,0)$. No matter how labels are assigned to them, a line can always separate them. Conversely, let's consider four points $(0,0)$, $(0,1)$, $(1,0)$, $(1,1)$ with label $[+,-,-,+]$, representing the "XOR" function. This function cannot be separated by a line. For $x \in \mathbb{R}$, then the VC dimension is 2 because you cannot separate $+-+$. In general for $x \in \mathbb{R}^d$, the VC dimension for a linear classifier is $d + 1$.

The **VC dimension** is usually **unrelated** to the **number of free parameters of a model**.
For all $f \in \mathcal{F}$, with probability at least $1 - \delta$, we have:

$$R(f) \leq R_{\text{emp}}(f) + \underbrace{\sqrt{\frac{h(\log(2n/h) + 1) - \log(\delta/4)}{n}}}_{\text{VC confidence}}$$

where

- $h = VC(\mathcal{F})$ is the VC dimension of the family F;
- δ is a tolerance parameter;
- n is the sample size.

We can read this results as:

With probability approaching 1, no matter what the unknown probability distribution, given more and more data, the expected error for the functions that ERM endorses at each stage eventually approaches the minimum value of expected error of the functions in $\mathcal{F}$ if and only if $\mathcal{F}$ has finite VC dimension. If VC dimension is equal to ∞, like in the KNN case, we have that $R_{emp}(f) = 0$ and so $R(f) \leq \infty$ which is meaningless.

Intuitively, since the second term $R_{\text{emp}} \dots$ is an upper bound, if it is small, $R(f)$ will be small too. This result is fundamental due to the fact that we have no knowledge about P, thus we cannot compute $R(f)$, but we can have a **bound** for it using the second quantity. Indirectly we want to **reduce the bound** so that at the end the resulting **risk** is **minimized**.

Structural Risk minimization. It is a general framework for choosing the best classifier. Empirical Risk Minimization only takes care of the *estimation* error (variance) but it is not concerned with the **approximation error** (bias). The optimal model is found by striking a balance between the empirical risk and the capacity of the function class $\mathcal{F}$ (ex: the VC dimension).
The basic idea of *Structural Risk Minimization* (SRM) is:

1. Construct a **nested** structure for family of function classes $\mathcal{F}_1 \subset \mathcal{F}_2 \subset \dots$ with **non-decreasing VC dimension** $(VC(\mathcal{F}_1) \leq VC(\mathcal{F}_2) \leq \dots)$
2. For **each class** $\mathcal{F}_i$, find the solution f_i that **minimizes** the **empirical risk**.
3. Choose the function class $\mathcal{F}_i$, and the corresponding solution f_i that minimizes the risk bound (= empirical risk + VC confidence)

Notice that this idea resembles the pruning approach, where we started with a large NN which was trained with back-propagation, and finally reduced. Here the reasoning is the opposite, we start with a small VC dimension and we augment it.

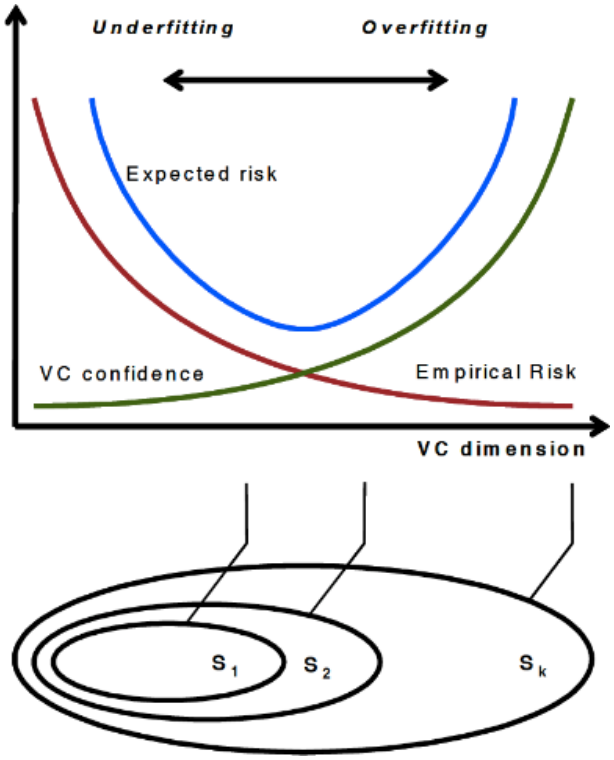


Figure 78: Structural risk minimization.

5 Support Vector Machines (SVMs)

5.1 Introduction

In this section we are going to deal with a very well known **supervised learning algorithm**, called **Support Vector Machine**. We will see that it can be written as an intuitive **optimization problem** and can be easily extended to work very well with non-linear patterns or in **high dimensional spaces**.

SVM belongs to the class of **discriminative classifiers** since its goal consists on **learning the class boundary between the two classes** y starting from features x. From now, we'll use $y \in \{-1, 1\}$ to denote the class labels. In a 2-dimensions feature space the decision boundary is represented by a straight line, while in general the decision boundary is represented by an **hyperplane**. An example of decision boundaries in a 2-dimensions feature space is represented in Picture 5.1.

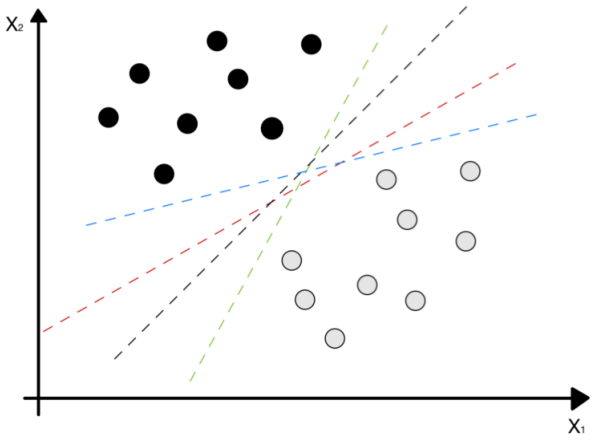


Figure 79: Decision boundaries in a 2-dimensions feature space

The data points that are closest to the decision boundary are called **support vectors**, while the distance between the support vectors along the perpendicular direction to the selected hyperplane is called **margin**: the hyperplane chosen by SVM is the one that maximize the margin. By choosing this particular hyperplane, the **misclassification risk** is minimized, since the **confidence** of the prediction of the model is stronger.
From a mathematical point of view, let's consider a 2-dimension feature space and let's assume labels are such that $y_i \in \{-1, 1\}$. A linear decision boundary B is defined by the equation:

$$w^T x + b = 0$$

, where w weights the features of x, so the objects above B are defined by $w^T x + b = k'$, where $k' > 0$, while the objects below B are define by $w^T xb = k''$, where $k'' < 0$. Let x_s and x_c be the positive and negative support vectors of B, we can then rescale w and b such that:

$$w^T x_s + b = 1$$

and

$$w^T x_c + b = -1$$

Let d_s and d_c be the distances between the support vectors and the decision boundary B, then by definition:

$$d_s = \frac{|w^T x_s + b|}{||w||} = \frac{|1|}{||w||} = \frac{1}{||w||}$$

$$d_c = \frac{|w^T x_c + b|}{||w||} = \frac{|-1|}{||w||} = \frac{1}{||w||}$$

Then, the margin d is defined as:

$$d = d_s + d_c = \frac{2}{||w||}$$

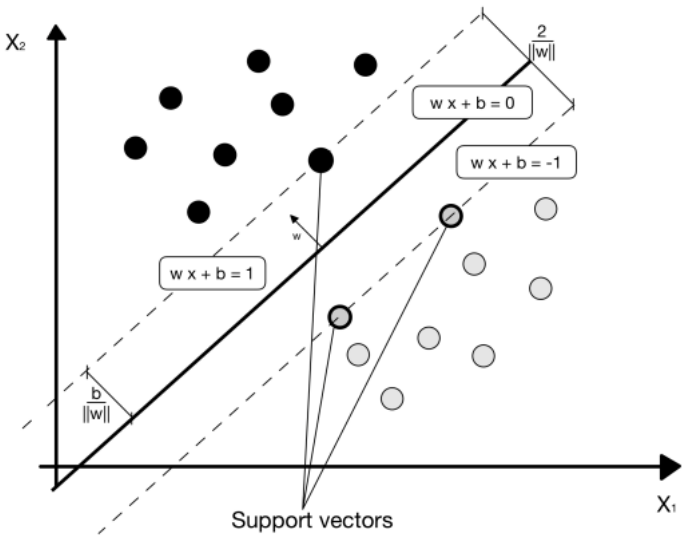


Figure 80: Geometric representation of SVM

Thus, given a training set $L = \{(x_1, y_1), .., (x_N, y_N)\}$, learning an SVM can be formulated as an optimization problem: indeed, the goal of SVM is to **maximize** $\frac{2}{||w||_2}$ or, equivalently, **minimize** $|||w||_2$. Finally, we can describe the SVM (binary) classification problem as:

$$\begin{aligned} \max_w \quad & \frac{2}{||w||} \\ \text{s.t.} \quad & y_i(w^T x_i + b) - 1 \geq 0 \quad \forall i = 1, .., N \end{aligned} \tag{1}$$

or, equivalently

$$\begin{aligned} \min_{w} \quad & \frac{1}{2}||w||^2 \\ \text{s.t.} \quad & y_i(w^T x_i + b) - 1 \geq 0 \quad \forall i = 1, .., N \end{aligned} \tag{2}$$

Since the objective functions in 1 and 1 are quadratic, and the constraints are linear in w and b, this is known to be a **convex optimization problem**, which means that there exists a unique minimum! This unique minimum corresponds to the **optimal margin classifier**.

5.2 Lagrangian and duality

5.2.1 Unconstrained optimization

Suppose we want to find the maximum of the following 2-dimensions function:

$$f(x, y) = 1 - x^2 - y^2$$

From calculus we know that the solution must be found among the points (x, y) where the gradient $\nabla f(x, y)$ vanishes, i.e. in the so-called **stationary points**:

$$\frac{\partial f(x, y)}{\partial x} = -2x = 0$$

and

$$\frac{\partial f(x, y)}{\partial y} = -2y = 0$$

In this sense, the solution is given by $x = 0$ and $y = 0$.

5.2.2 Constrained optimization

Suppose now that the points (x, y) have to lie on the straight line of equation $x + y = 1$, i.e. we want to solve the following constrained optimization problem:

$$\begin{aligned} \max \quad & 1 - x^2 - y^2 \\ \text{s.t.} \quad & x + y - 1 = 0 \end{aligned}$$

Our goal is now to define a method for transforming the constrained optimization problem into an unconstrained one. To this end, we define a new function $L(x, y, \lambda)$, called the **Lagrangian** as follows:

$$L(x, y, \lambda) = 1 - x^2 - y^2 + \lambda(x + y - 1)$$

, where $\lambda \neq 0$ is called a **Lagrangian multiplier**, and looks for points (x, y, λ) where the gradient ∇L vanishes. Notice that in L we now have as many new variables as the constraints we have. If we apply this method to example, we get the following system of linear equations:

$$\frac{\partial L(x,y,\lambda)}{\partial x} = -2x + \lambda = 0$$
$$\frac{\partial L(x,y,\lambda)}{\partial y} = -2y + \lambda = 0$$
$$\frac{\partial L(x,y,\lambda)}{\partial \lambda} = x + y - 1 = 0$$

, from which we get the solution $x = y = \frac{1}{2}$

5.2.3 General case

More generally, given the following optimization problem:

$$\begin{aligned} \max \quad & f(x) \\ \text{s.t.} \quad & g_1(x) \geq 0..g_m(x) \geq 0 \\ & h_1(x) = 0..h_n(x) = 0 \end{aligned}$$

, where we have m inequality constraints and n equality constraints, the Lagrangian is defined as:

$$L(x, \Lambda, M) = f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{n} \mu_j h_j(x)$$

, where $\Lambda = (\lambda_1, .., \lambda_m)^T$ and $M = (\mu_1, .., \mu_n)^T$ are vectors of Lagrange multipliers corresponding to inequality and equality constraints, respectively.
In this case we need to impose the conditions $\lambda_i \geq 0$ and $\lambda_i g_i(x) = 0$ for all $i = 1, .., m$
The **intuition** behind the **equality constraints** is the following one: at any point x in the constraint surface, $\nabla g(x)$ is normal to the surface, from the properties of the gradient. Then, if x is also a maximizer of f, $\nabla f(x)$ must be orthogonal to the surface too (otherwise we could increase the value of f with another point), so we have that:

$$\nabla f(x) = -\lambda \nabla g(x)$$

, with $\lambda \neq 0$.

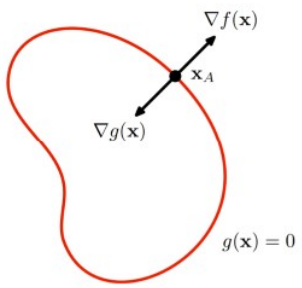


Figure 81: Equality constraints

The **intuition** behind the **inequality constraints** is the following one. We have two cases:

- The solution is in the interior, i.e. $g(x) > 0$: in this case the stationary point condition implies $\nabla f(x) = 0$, which corresponds to the case $\lambda = 0$;
- The solution is on the boundary, i.e. $g(x) = 0$: this is analogous to the previous case, but this time the sign of λ is crucial: hence $\nabla f(x) = -\lambda \nabla g(x)$, with $\lambda > 0$.

For either of these cases we have $\lambda g(x) = 0$

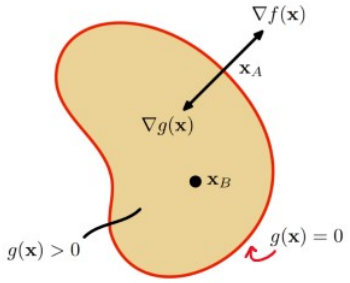


Figure 82: Inequality constraints

Notice that if we want to minimize, rather than maximize, $f(x)$, the Lagrangian multipliers have to be non-positive, or simply change the sign of the corresponding term in the Lagrangian.

5.2.4 Duality

Consider the following optimization problem with inequality constraints, called the **primal**:

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & g_1(x) \geq 0..g_m(x) \geq 0 \end{aligned}$$

with optimal value p^*, and consider its Lagrangian:

$$L(x, \lambda_1, .., \lambda_m) = f(x) - \sum_{i=1}^{m} \lambda_i g_i(x)$$

Notice that the Lagrangian has a minus sign since the problem is a minimization.
Given $\lambda_1, .., \lambda_m \geq 0$, we define the (Lagrangian) **dual function** as:

$$\phi(\lambda_1, .., \lambda_m) = \inf_x L(x, \lambda_1, .., \lambda_m)$$

It's easy to see that $\phi(\lambda_1, .., \lambda_m) \leq p*$, i.e. the optimal value of the previous problem is an upper bound of this second problem.
The problem

$$\begin{aligned} \max \quad & \phi(\lambda_1, .., \lambda_m) \\ \text{s.t.} \quad & \lambda_1, .., \lambda_m \geq 0 \end{aligned}$$

is called the (Lagrangian) **dual** of problem 5.2.4.

Weak duality: if p^* is a solution of the primal and d^* is a solution of its dual, then:

$$d^* \leq p^*$$

The quantity $d^* - p^* \leq 0$ is called **duality gap**.

Strong duality: if the function f of the primal is convex (and so are all $-g_i$), then

$$p^* = d^*$$

, i.e. the solution of the dual (which is simpler) is the same as the solution of the primal.

Wolfe duality: the **Wolfe dual** is defined as:

$$\begin{aligned} \max_{x,\Lambda} \quad & L(x, \Lambda) \\ \text{s.t.} \quad & \nabla_x L(x, \Lambda) = 0 \\ & \Lambda \geq 0 \end{aligned}$$

, where $\Lambda = (\lambda_1, .., \lambda_m)$.
Assume that functions f and $-g_i$ are convex (and continuously differentiable): if x^* is a solution of the **primal problem**, then there exists a **vector of Lagrangian multipliers** Λ^* s.t. (x^*, Λ^*) is a **solution** of the **Wolfe dual**, and the duality gap is 0.

5.2.5 Dual representation of SVM

In order to solve constrained problem

$$\begin{aligned} \min_{w} \quad & \frac{1}{2}||w||^2 \\ \text{s.t.} \quad & y_i(w^T x_i + b) - 1 \geq 0 \quad \forall i = 1, .., N \end{aligned} \tag{1}$$

we introduce the concept of **Lagrange multipliers**. Using the **Lagrange** function with N Lagrange multipliers $\Lambda = (\lambda_1, \ldots, \lambda_N)$ (one for each constraint) it is possible to rewrite the correspondent optimization problem, with parameters w and b, into an identical one but with parameters $(\lambda_1, \ldots, \lambda_N)$. Taking advantage of the **dual representation**, that allows to convert an original min/max problem into another one which is equivalent but with max/min formulation, we can write the **new optimization problem**as follows:

$$L(w, b, \Lambda) = \frac{1}{2}||w||^2 - \underbrace{\sum_{i=1}^{N} \lambda_i [y_i(w^T x_i + b) - 1]}_{\text{Sum of constraints}} \tag{2}$$

, where $\Lambda = (\lambda_1, .., \lambda_N)$ is the vector if Lagrange multipliers.
The goal now is to find a **function parameterized** only by **Lagrangian multipliers**. Setting the derivatives of $L(w, b, \Lambda)$ to zero we get:

$$\frac{\partial L(w, b, \Lambda)}{\partial w} = w - \sum_{i=1}^{N} \lambda_i y_i x_i = 0 \qquad \Longrightarrow \qquad w = \sum_{i=1}^{N} \lambda_i y_i x_i$$

$$\frac{\partial L(w,b,\Lambda)}{\partial b} = \sum_{i=1}^{N} \lambda_i y_i = 0$$

Eliminating w and b from $L(w,b,\Lambda)$ using these conditions we obtain a new formulation of the optimization problem expressed with only Lagrangian multipliers:

$$\begin{aligned} \max \quad & L_D(\lambda_1,\ldots,\lambda_N) = \sum_{i=1}^{N} \lambda_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \lambda_i\lambda_j y_i y_j x_i^T x_j \\ \text{s.t.} \quad & \sum_{i=1}^{N} \lambda_i y_i = 0 \qquad \lambda_i \geq 0, \forall i = 1,\ldots,N \end{aligned}$$

On the one hand, in this case we have a number of variables which is equal to the number of examples in the training set, i.e. we have much **more variables**. However, it can be noticed that the **training vectors** only appear as *dot products*, and the advantage of using this formulation is that only **support vectors** will have **Lagrange multipliers** such that $\lambda_i > 0$ and the **others** will be essentially **equal to zero** (sparse solution). The **SVM complexity** is only given by the **support vectors** (which are much less than the number of examples in the training set). Now the **maximum margin hyperplane** is given by:

$$\sum_{i=1}^{m} y_i \lambda_i x_i^T x + b = 0$$

If $\Lambda = (\lambda_1,\ldots,\lambda_N)$ is the solution of the dual optimization problem, then:

- The **weight vector** of the maximum margin hyperplane is: $w = \sum_{i=1}^{N} y_i \lambda_i x_i$
- The corresponding **discriminant function** (separating hyperplane) is:

$$f(x) = w^T x + b = \sum_{i=1}^{N} y_i \lambda_i x_i^T x_i + b$$

- The **linear SVM classifier** $g : \mathbb{R}^n \to \{-1, 1\}$ is:

$$g(x) = \text{sign}(w^T x + b) = \text{sign}(\sum_{i=1}^{N} y_i \lambda_i x_i^T x_i + b)$$

For support vectors we have:

$$y_i(\sum_{i=1}^{N} y_i \lambda_i x_i^T x_i + b) = \gamma$$

, and for simplicity we consider $\gamma = 1$. Since SVM is affected only by support vectors we can derive:

$$b = \frac{1}{|SV|} \sum_{i \in SV} (y_i - \sum_{j=1}^{N} y_j \lambda_j x_j^T x_i)$$

where **SV** is the **set of support vectors**.

5.2.6 SVM error function

The generic loss function adopted for training a support vector machine is the **Hinge loss function**.

$$L_{\text{hinge}} = \max\{0, 1 - y_i f(x_i)\}$$

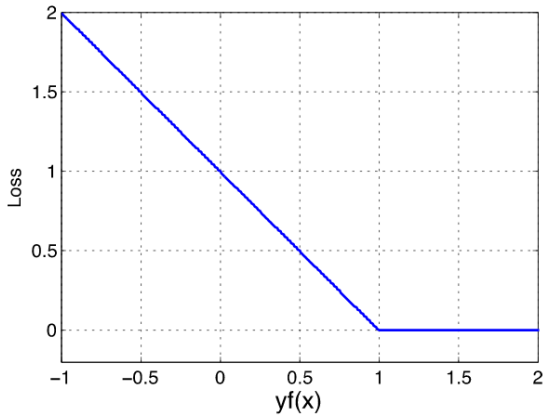


Figure 83: Hinge loss function.

$$E = \sum_{i=1}^{N} \max\{0, 1 - y_i f(x_i)\} + \frac{1}{2} \sum_{j=1}^{d} w_j^2$$

, where

- $\sum_{i=1}^{N} \max\{0, 1 - y_i f(x_i)\}$ represents the hinge loss function over all data points;
- $\frac{1}{2} \sum_{j=1}^{d} w_j^2$ is proportional to the inverse of the margin.

Notice that the hinge loss function is not differentiable on 1, we can't apply the gradient descent procedure. The solution of this problem consists on using **quadratic programming (QP)** algorithms, since E is a quadratic function with linear constraints.

5.2.7 SVMs and the VC dimension

Theorem (Vapnik) Consider **hyperplanes** $w^T x + b = 0$ in canonical form, that is such that:

$$\min_{1 \leq i \leq N} |w^T x_i + b| = \gamma = 1$$

Then the **set of decision functions** $g(x) = sgn(w^T x + b)$ (i.e. of SV classifiers) that satisfy the **constraint** $||w|| < \gamma$ has a **VC dimension** h satisfying:

$$h \leq R^2 \gamma^2$$

where R is the **smallest radius** of the sphere around the origin containing all the **training points**. Note that dropping the condition $||w|| < \gamma$ leads to a VC dimension equal to

$n+1$. Hence, the constraint allows us to work in high-dimension spaces.
From the previous theorem and from:

$$R(f) \leq R_{\text{emp}}(f) + \sqrt{\frac{h(\log(\frac{2n}{h}) + 1) - \log(\frac{\sigma}{4})}{n}}$$

we have:

- By **maximizing the margin**, or equivalent by **minimizing** $||w||$, we are in fact **minimizing** the **VC dimension** of the SVM;
- The **minimization of the expected** risk depends on both **minimizing** the **empirical risk** and the **confidence interval**;
- The **confidence interval** depends mainly on the **ratio** $\frac{h}{n}$;
- The **SVM algorithm minimizes** both the **empirical risk** and the **confidence interval**;
- The **SVM** directly implements the **structural risk minimization principle**.

5.3 How to manage outliers: soft margins

One of the problems of SVMs is to make decisions in presence of **outliers**.

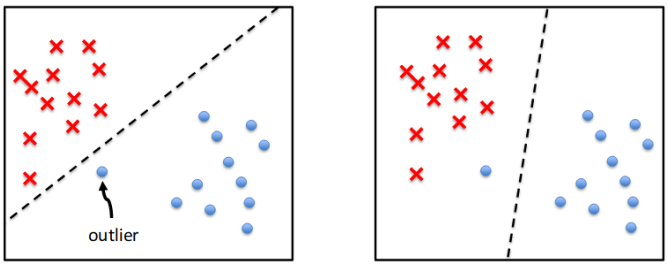


Figure 84: Outliers effects on an SVM.

The previous image shows the effect that a single outlier can have. We can make only one of the following choices: **correctly classify** the outlier, thus improving the **accuracy** of the classifier and produce a **smaller margin**, or leave the outlier **mis-classified**, thus diminishing the accuracy but resulting in a **larger margin**. The strategy which is commonly adopted is the **second one**, as the first one would find **decision boundaries** that work well on the training set, but do **not provide optimal performance during testing**.
In order to reduce the effects of mis-classification we adopt some **slack variables**, useful for allowing some errors in the boundary.

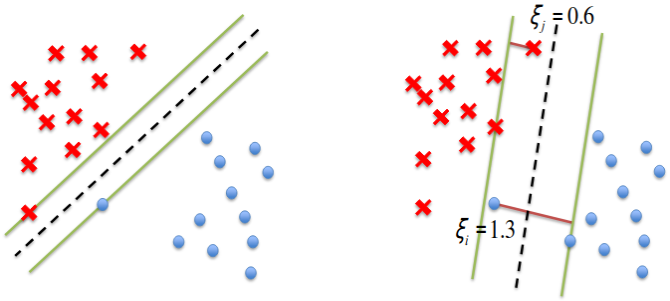


Figure 85: Outliers effects on an SVM with slack variables.

These **variables** indicate **how** much we can **violate** the **constraints** of SVM, and a slack variable is added for each of the points in the dataset.
Now the optimization problem can be reformulated as follows:

$$\begin{aligned} \min \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{N}\xi_i \\ \text{s.t.} \quad & y_i(w^T x_i - b) \geq 1 - \xi_i \\ \text{s.t.} \quad & \xi_i \geq 0 \qquad i = 1, \ldots, N \end{aligned}$$

The only parameter C controls the **tradeoff** between the **accuracy** with respect to the training data and the **maximization** of the **margin**. It can be interpreted also as a **regularization term**:

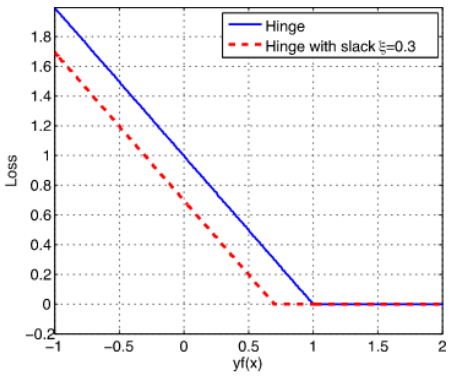
- **small** C allows constraints to be easily ignored *(large margin)*.
- **large** C makes constraints hard to ignore *(narrow margin)*.
- $C = \infty$ enforces **all constraints** *(hard margin)*.

The **dual representation** of the problem can be reformulated as follows:

$$\begin{aligned} \max \quad & L_D(\lambda_1, \ldots, \lambda_N) = \sum_{i=1}^{N}\lambda_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\lambda_i\lambda_j y_i y_j x_i^T x_j \\ \text{s.t.} \quad & \sum_{i=1}^{N}\lambda_i y_i = 0 \qquad \forall i = 1, \ldots, N \\ \text{s.t.} \quad & 0 \leq \lambda_i \leq C \qquad \forall i = 1, \ldots, N \end{aligned}$$

The hyperplanes whose weight vectors solve this quadratic optimization problem are called the **soft margin hyperplanes**. The soft-margin optimization problem is equivalent to that of the maximum margin hyperplanes with the additional constraint $\lambda_i \leq C$ (box constraints). This **approach limits** the effect of the **outliers** (for which λ_i tends to be large).

$$L_{hinge} = \max\left\{0, 1 - y_i f(\mathbf{x}_i) - \xi_i\right\},$$



$$E = \sum_{i=1}^{N} \max\left\{0, 1 - y_i f(\mathbf{x}_i) - \xi_i\right\} + \frac{1}{2}\sum_{j=1}^{d} w_j^2 + C\sum_{i=1}^{N} \xi_i$$

slack

penalty for using slack

Figure 86: Hinge loss function with slack variables.

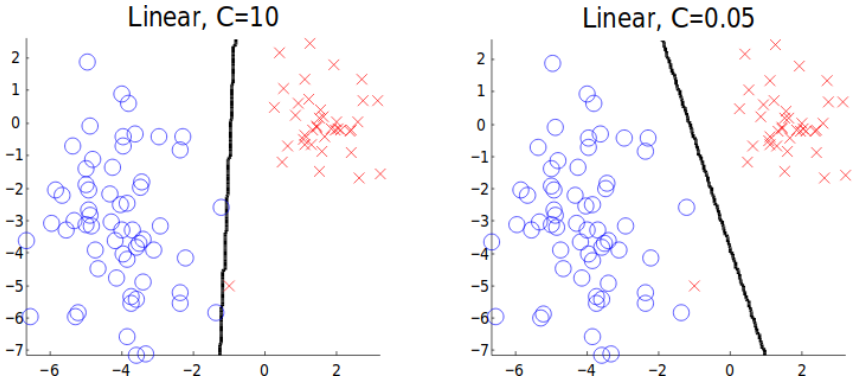


Figure 87: Effect of C on the decision boundary.

5.4 Nonlinear SVM's: Kernel trick

Thus far we worked with the assumption that the space is **linearly separable**, but SVMs allow the usage of a strategy, called **kernel trick**, for learning a possible separating hyperplane in a **new space**. As a matter of fact, in some cases it could be interesting to try and classify points in a **transformation of the original space**. The classic situation in which we may apply the kernel trick is when **data points** are **not linearly separable** in the original space.

The idea is to define a function $\phi(x)$ that applies a **mapping** of a **feature vector** to **another** one. The SVM algorithm, instead of considering vector x, learns using the transformed vector $\phi(x)$. A **kernel** function is nothing but an **inner product** between feature mappings of ϕ:

$$K(x, z) = \phi(x)^T \phi(z)$$

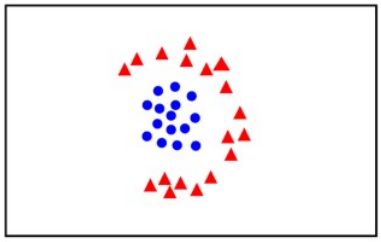


Figure 88: Non-linear problem

Example 5.1. The following Picture represents an example of possible mapping of the data points into a new space in order to make them linearly separable.

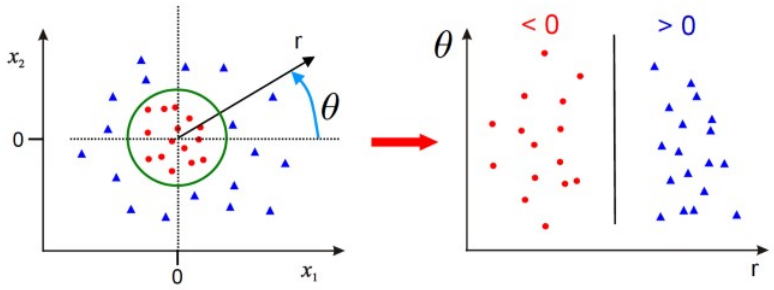


Figure 89: Example of mapping

As we can see, data is now represented in polar coordinates (each point is defined by the radius r and the angle θ), so the points are now linearly separable. In this case $\phi : (x_1, x_2) \in \mathbb{R}^2 \to (r, \theta) \in \mathbb{R}^2$, so in this case we did not project the feature in a larger dimensionality vector.

Cover's Theorem *A complex pattern-classification problem cast in a high-dimensional space non-linearly is more likely to be linearly separable than in a low-dimension space.*

In other words, this theorem states that if we map the points of a problem which is not linearly separable into a higher dimensional space, then the problem is more likely to be linearly separable. The **power** of **SVM's** resides in the fact that they represent a **robust** and **efficient implementation** of Cover's **theorem**.
In general, nonlinear SVM's operate in two stages:

1. Perform a (typically implicit) **non-linear mapping** ϕ of the feature vector x onto a high-dimensional space that is hidden from the inputs or the outputs. This represents the most difficult step;
2. Construct an **optimal separating hyperplane** using SVM's in the high-dimensional space.

We recall that in the dual representation of SVM's the inputs appears only in a **dot-product form**, i.e.

$$\begin{aligned}
\max \quad & L_D(\lambda_1, \ldots, \lambda_N) = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j y_i y_j x_i^T x_j \\
\text{s.t.} \quad & \sum_{i=1}^{N} \lambda_i y_i = 0 \qquad \forall i = 1, \ldots, N \\
\text{s.t.} \quad & 0 \leq \lambda_i \leq C \qquad \forall i = 1, \ldots, N
\end{aligned}$$

and the discriminant function obtained from the solution is:

$$f(x) = \sum_{i=1}^{N} y_i \lambda_i x_i^T x + b$$

Now we can **replace** the simple **inner product** with a **kernel function** in order to learn in a different feature space, which in some cases could be more efficient. There is a restriction on the function K: it must satisfy the following **property** (called Mercer's condition) in order to be considered a **valid kernel**:

$$K(x, z) = \phi(x)^T \phi(z) \qquad \forall x, z \in S$$

In this sense, suppose we first **mapped** the data to some other (possibly infinite dimensional) Euclidean space, using a mapping:

$$x \to \phi(x) \qquad K(x, y) = \phi(x)^T \phi(y)$$

$$\begin{aligned}
\max \quad & L_D(\lambda_1, \ldots, \lambda_N) = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j y_i y_j K(x_i, x_j) \\
\text{s.t.} \quad & \sum_{i=1}^{N} \lambda_i y_i = 0 \qquad \forall i = 1, \ldots, N \\
\text{s.t.} \quad & 0 \leq \lambda_i \leq C \qquad \forall i = 1, \ldots, N
\end{aligned}$$

Now, the **discriminant function** is:

$$f(x) = \sum_{i=1}^{N} y_i \lambda_i K(x_i, x) + b$$

Note that now is **not necessary** to compute $\phi(x)$.
There exist many kernels, for instance:

- **Linear**: $K(x_i, x_j) = x_i^T x_j$, which performs an identity mapping;
- **Polynomial kernel**: $K(x_i, x_j) = (1 + x_i^T x_j)^d$, for any $d > 0$;
- **Gaussian kernel** (RBF): $K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$, for any $\sigma > 0$.

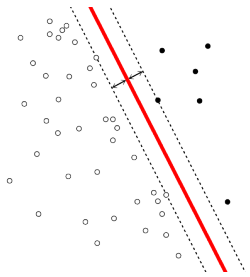


Figure 90: Linear kernel

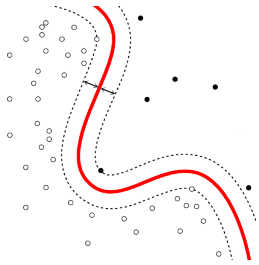


Figure 91: Polynomial kernel

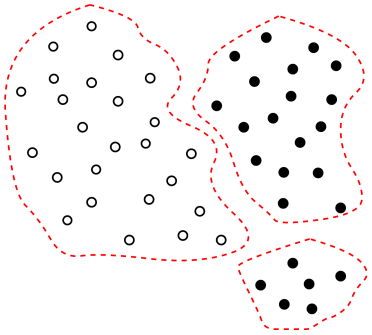


Figure 92: Gaussian kernel

5.5 Multi-class problems

Thus far we have only discussed about the application of an SVM using **two labels** $y \in \{-1, 1\}$. In real cases, however, we can have more than two classes and we might need to develop an SVM capable of assigning input vectors to one in K **classes**. In other words, we have to find a decision rule that divides the input space into K **decision regions** separated by decision boundaries.

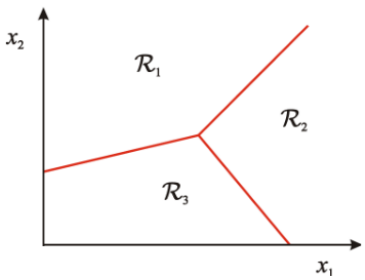


Figure 93: Decision boundaries for 3 classes.

We can apply two distinct strategies:

- **One-vs-rest classifiers**: train $K - 1$ classifiers, **each** of which solves a **two-class problem** of separating points in a particular class from points not in that class.
- **One-vs-one classifiers**, train $K(K - 1)/2$ **binary classifiers**, one for every possible pair of classes.

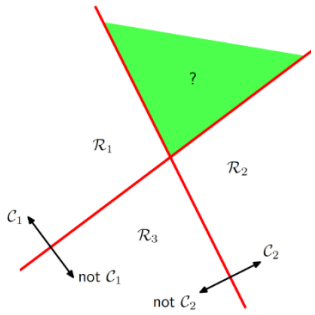


Figure 94: One-vs-the-rest classifiers.

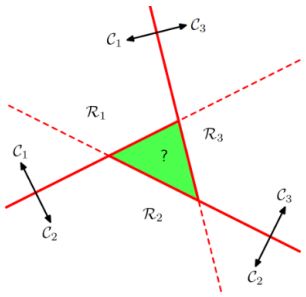


Figure 95: One-vs-one classifier.

Note that in the first case the green area denotes a region in which the points are both in C_1 and in C_2, while in the second case the points in the green area are in C_1, C_2 and C_3 at the same time. Thus, in both cases we have contradictory results. The **classical approach** consists on training K **one-vs-rest classifiers** and then the classification is done choosing the **class** with the "**most positive**" score.

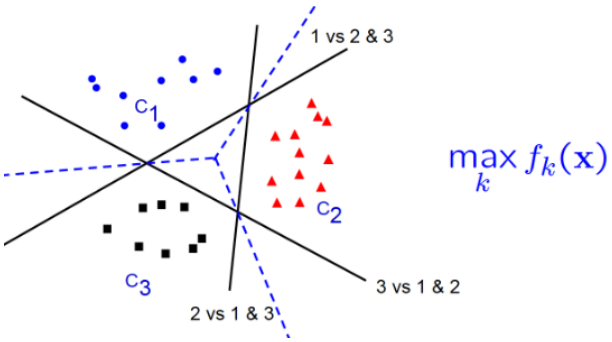


Figure 96: Decision boundaries on typical approaches.

5.6 Advantages and disadvantages

Among the **advantages** we can find:

- SVM works relatively **well** when there is a **clear margin** of separation between classes;
- SVM is **more effective** in **high dimensional spaces** and is relatively memory efficient;
- SVM is **effective** in cases where the **dimensions are greater than the number of samples**.

On the other hand, the **disadvantages** are:

- SVM algorithm is **not suitable** for **large data sets**;
- SVM does **not perform very well** when the data set has more **noise** i.e. target classes are overlapping. In cases where the number of features for each data point exceeds the number of training data samples, the SVM will underperform.

- As the support vector classifier works by putting data points, above and below the classifying hyperplane there is **no probabilistic explanation** for the classification.

6 Clustering

The **classical clustering** problem starts with

- A set of n objects;
- A $n \times n$ matrix A of pairwise similarities that gives us an edge-weighted graph G.

, and the goal is to **partition** the vertices of G into **maximally homogeneous groups (clusters)**. Usually we make the following assumptions:

- The **similarity metric** is **symmetric**, i.e. $\text{sim}(a, b) = \text{sim}(b, a)$. However, this is not always the case: if, for example, we consider the case of computing a similarity between two documents represented using *bag of words* (each document is represented as a probability distribution of its terms), then the *KL divergence* can be used for computing the similarity, but we've seen that this measure is not symmetric;
- We only consider **pairwise similarities**, i.e. similarity between two objects. Notice that there are situations in which we may compute the similarity between more that 2 objects, e.g. with **tensors** or **hypergraphs**.
- The graph G is an undirected graph.

Figure 97: The "classical" clustering problem.

Clustering problems abound in many areas of CS, e.g. image processing and CV, IR, document analysis, data mining etc.. If we consider, for example, the image segmentation problem, it's easy to see that it "simply" consists in clustering similar pixels of an image into coherent regions.

6.1 Feature-based clustering algorithm: K-means

K-means is an iterative clustering algorithm that relies on the following assumptions:

- It is provided as input with **feature vectors**, and for this reason we refer to K-means as a **feature-based** (or **central**) **clustering** algorithm, i.e. it receives in input points in a high-dimensional feature space. The other approach is represented by the **graph-based** (or **pairwise**) **clustering** algorithm, in which we are given either a **similarity matrix** or a **graph** in which the weights represent the similarity between the entities. In this latter case we do not make any assumption about the representations of the objects;

- The **number of clusters** is known in advance (in many applications this is a problem).

The algorithm follows these steps:

- **Initialize:** pick K random points as cluster centers.

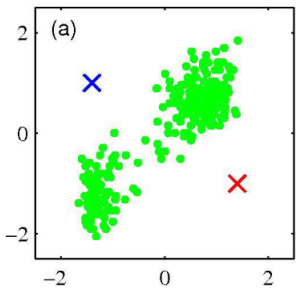


Figure 98: Initialization with $K = 2$.

- **Alternate:**
  1. **Assign** data points to **closest cluster center**.
  2. **Change the cluster center** to the **average** of its assigned points.

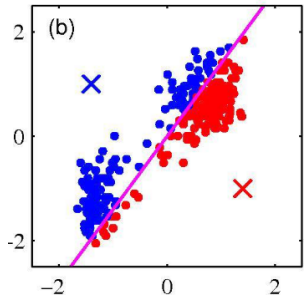


Figure 99: Iterative step 1.

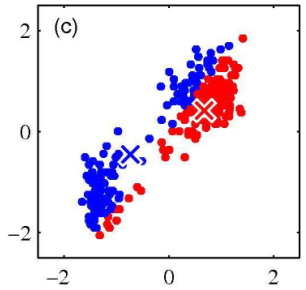


Figure 100: Iterative step 2.

- **Stop:** when no points' assignments change.

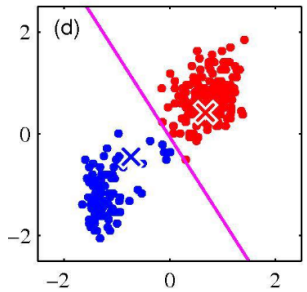


Figure 101: Repeat until convergence.

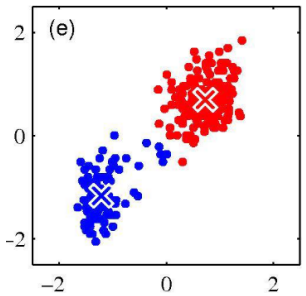


Figure 102: Final output.

Advantages of K-means.

- It is a **simple algorithm**;
- It is guaranteed to **converge** in a **finite number of steps**;
- It **minimizes** an **objective function** (i.e. it **maximizes** the **compactness** of clusters):

$$\sum_{i \in \text{clusters}} \left\{ \sum_{j \in \text{elements of i-th cluster}} ||x_j - \mu_i||^2 \right\}$$

  where μ_i is the center of cluster i;
- It **assigns** data points to closest cluster center in $O(Kn)$ and it **changes** the cluster center to the average of its points in $O(n)$, so it is an **efficient** algorithm.

Disadvantages of K-means

- It converges to a **local minimum** of the error function, i.e. we have no theoretical guarantees that the algorithm will converge to a global minimum, and that's the reason why we may run the algorithm several times;
- It needs to pick K **initial points**, hence proving to be very **sensitive to the initialization step**;
- It is also very sensitive to the **outliers**, which are not known in advance;
- It only finds **spherical clusters**, due to the objective function it minimizes. In this sense, this algorithm does not work when we have non-convex clusters;
- It works with **feature vectors**, which sometimes are not easy to obtain.

6.2 Eigenvector-based clustering

As we introduced before, the other possible approach for unsupervised learning is represented by the **graph-based clustering**, in which the input is represented either by a **similarity matrix** or a **graph**.

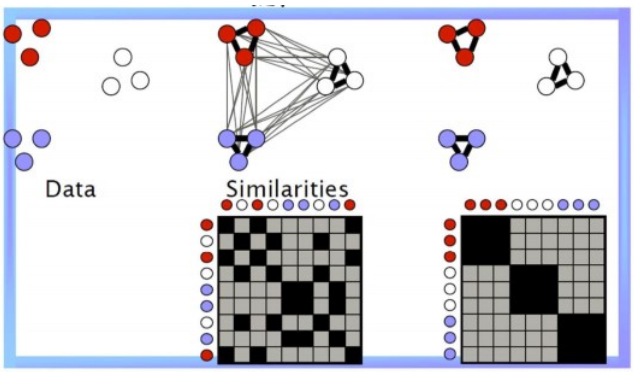


Figure 103: Graph and matrices for clustering

It is important to underline the fact that the two representations are equivalent, but they have different characteristics:

- If we consider the **similarity matrix**, then eivenvalues and eigenvectors play a crucial role in the clustering problem, since they provide an information of the matrix which is independent of the permutations of the matrix. In this sense, despite the fact that there may exist many matrices representing the same graph, the resulting eigenpairs remain the same;
- If we consider the **graph** representation, then the notion of **cut** plays a crucial role in clustering: this problem, indeed, reduces in finding the cut(s) of the graph which separate the object of different classes from the others, i.e. that maximize the **intra-class similarity** and minimize the **inter-class similarity**.

We now focus on the clustering problem that deals with a **similarity matrix**.

6.2.1 Eigenvalues and eigenvectors

Before defining the clustering problem as a **eigenvector-based** problem, we make a little digression on eigenpairs. We first focus on two cases:

- Suppose we have a **symmetric matrix** A, then all the **eigenvalues** are **real**, which means we can consider an order between the eigenvalues. This property introduces a very strong assumption of the **spectral graph theory**, i.e. that the input matrix must be symmetric;
- Suppose we have a **non symmetric matrix** A, then A must be symmetrized, i.e. we must build a symmetrix matrix A' as:

$$A' = \frac{1}{2}(A + A^T)$$

Now, suppose that the matrix A is symmetric, we can compute the largest eigenvalue λ_{MAX} of A by solving one of the following two maximization problem:

$$\lambda_{\text{MAX}} = \max_x \quad x^T A x \quad \text{s.t.} \quad x^T x = 1 \tag{1}$$

or

$$\lambda_{\text{MAX}} = \max_x \quad \frac{x^T A x}{x^T x} \quad \text{s.t.} \quad x \in \mathbb{R}^n \tag{2}$$

, where $\frac{x^T A x}{x^T x}$ is defined as **Rayleigh quotient**. Notice that 1 is a **constrained optimization problem**, while 2 is an **unconstrained optimization problem**.
On the other hand, the general eigenvector/eigenvalue problem is defined by finding the λ s.t. $Ax = \lambda I x$, where I represents the **identity matrix**: if we have another matrix B, then the solution becomes $Ax = \lambda B x$, and the solution of finding λ_{MAX} is:

$$\max_{\lambda_{\text{MAX}}} \quad \frac{x^T A x}{x^T B x} \quad \text{s.t.} \quad x \in \mathbb{R}^n \tag{3}$$

6.2.2 Problem definition

Let us represent a **cluster** using a **vector** x whose i-th entry captures the participation of node i in that cluster:

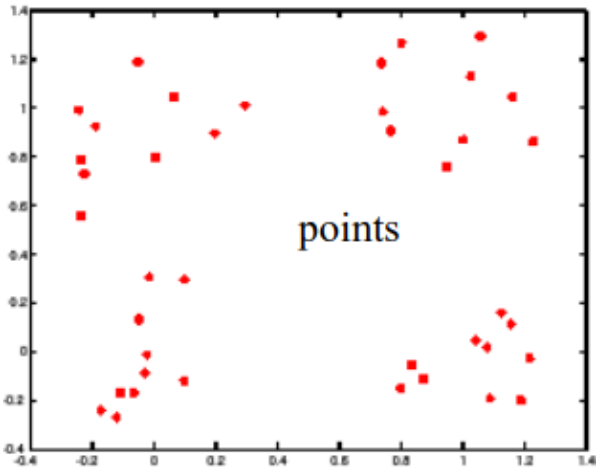
$$x_i : \begin{cases} \neq 0 \text{ if } i \in C \\ = 0 \text{ if } i \notin C \end{cases}$$

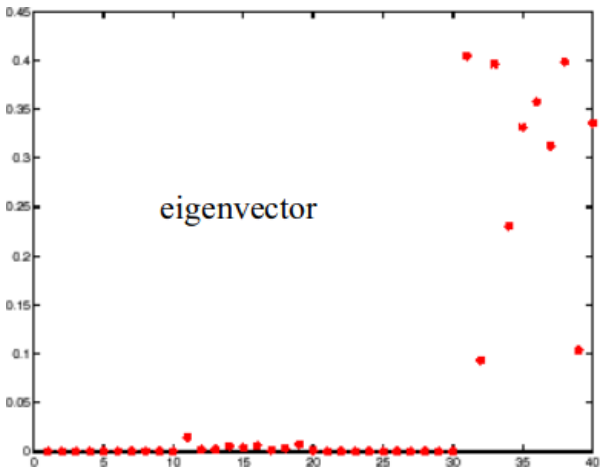
If a node does not participate in a cluster, the corresponding entry is zero. We also impose the **restriction** that $x^T x = 1$ in order to avoid the trivial solution $x = 0$.
Thus, we want to maximize:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} x_i x_j = x^T A x$$

which measures the **cluster's cohesiveness**. In this sense, out goal is to maximize the internal similarity of the points within the cluster. As we introduced in the previous section, this is an **eigenvalue problem**, which consists in choosing the **eigenvector** of A corresponding to the **largest eigenvalue**.

From the following image we can see that it could be possible to find clusters by visual inspection of the eigenvectors.





However, solving the previous problem only allows to find a **single cluster**, so in case we must extract **more than two clusters** from the eigenvectors, we can consider one of the following strategies:

1. Recursively split each side to get a tree, continuing till the eigenvalues are too small;
2. Use not only the largest eigenvalue, but also the others. A powerful result in linear algebra says that the second largest eigenvalue (i.e. the eigenvalue associated to the second largest eigenvector) can be obtained by considering

$$\max \quad \frac{x^T A x}{x^T x} \quad \text{s.t.} \quad x \in \mathbb{R}^n \text{and } x \perp x_{\max} \tag{4}$$

, where $x_{\max}$ represents the eigenvector associated to the largest eigenvalue.

6.2.3 Clustering by eigenvectors : algorithm

The algorithm that builds the clusters through the eigenvectors performs the following steps:

1. Construct (or take as input) the **affinity matrix** A;
2. Compute the **eigenvalues** and **eigenvectors** of A;
3. Repeat until there are sufficient clusters:
4. Take the **eigenvector** corresponding to the **largest** unprocessed **eigenvalue**;
5. **Zero** all the components corresponding to **elements** that have already **been clustered**;
6. **Threshold** the remaining components to determine which elements belong to this cluster;
7. If all elements have been accounted for, there are sufficient clusters;

6.3 Graph-based clustering algorithm

If we consider the **graph** representation we have that:

- A **node** represents each of the **pixels**;
- An **edge** between every pair of pixels (or every pair of "sufficiently close" pixels), which is weighted according to the affinity or **similarity** of the two nodes.

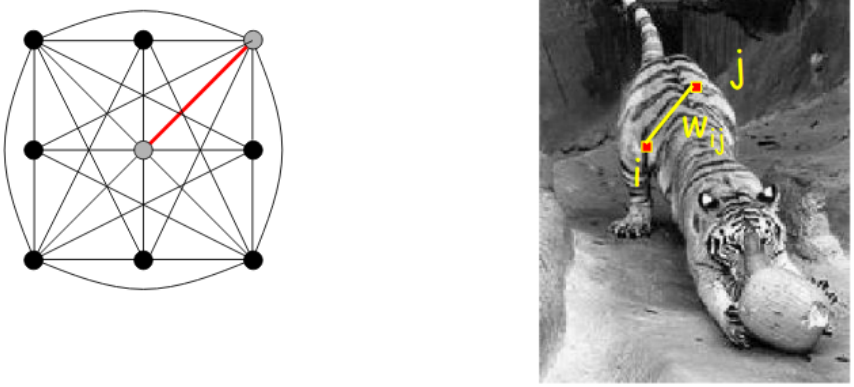


Figure 104: Image as a graph.

If we suppose to represent each **pixel** with a **feature vector** x and to define a **distance function** appropriate for this feature representation, then we can **convert** the **distance** between two feature vectors into an **affinity** with the help of a **Gaussian kernel**:

$$exp\left(-\frac{1}{2\sigma^2}dist(x_i, x_j)^2\right)$$

Notice that we can exploit this kernel to transform a feature-based dataset into a graph-based one. We can notice that the **similarity** of two data points is **inversely proportional** to their **distance**, and we also underline the importance of the **scale** parameter

σ: the **smaller** σ, the **more rigid** will be the clustering algorithm in grouping together only nearby points; on the other hand, the **larger** σ, the more the algorithm will **group** together **far-away points**.
We now focus on the clustering as a **graph partitioning problem**. Let $G = (V, E, w)$ be an undirected weighted graph, i.e. the similarity matrix is symmetric, and given a partition (or cut) (C_1, C_2) of the graph G, then the following quantity can be defined:

$$cut(C_1, C_2) = \sum_{i \in C_1} \sum_{j \in C_2} w(i, j)$$

In Picture 6.3, the cut of the two sub graphs is given by the sum of the weights of the red edges.

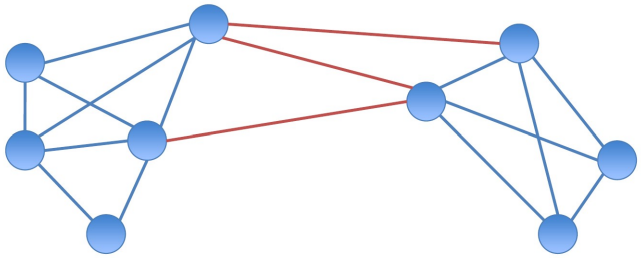


Figure 105: Cut in a graph

Minimum cut Given this quantity, we can easily connect the problem of determine the clusters of a graph to the one of finding a cut that maximizes the intra-cluster similarity and minimizes the inter-class similarity of the points contained in the two partitions that are formed. More specifically, finding the clusters in a graph is equal to solve the so called **minimum cut problem**. This problem tries to find the cut that minimizes the quantity $cut(C_1, C_2)$ among all the possible cuts, i.e. partitions, (C_1, C_2).
Despite the fact that the number of cuts in a graph grows exponentially with the number of nodes, an important property of the *minimum cut* problem is that it is solvable in polynomial time. However, a quite important disadvantage of this problem is that it favors highly unbalanced clusters, in particular the ones composed by single vertices, as represented in Picture 6.3.

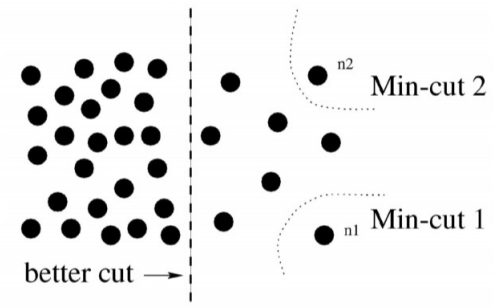


Figure 106: Disadvantage of MinCut problem: isolated vertices form a cluster

This problem derives from the fact that it takes into account only the intra-cluster similarity, resulting in this kind of undesired partitions.

Normalized cut The other possible approach when dealing with graph partitioning, and the one adopted in this project, is represented by solving the **normalized cut** problem. Before discussing the details of this problem, we introduce some important metrics:

- the *degree* of a node is defined as $\deg(i) = \sum_j w_{i,j}$, i.e. it is represented by the sum of the values in the i-th row of the matrix representing the graph;
- the *volume* of a set of nodes is defined as $vol(A) = \sum_{i \in A} d_i$, where $A \subseteq V$, i.e. it is represented by the sum of the degrees of the nodes in the set (sum of multiple rows).

The idea of *normalized cut* is to overcome the limits of *minimum cut* by normalizing the previous quantity by a measure (the volume) that allows to combine both the intra-cluster similarity and the inter-cluster similarity. More specifically, for the *normalized cut* problem, the quantity that has to be minimized is the following one:

$$Ncut(C_1, C_2) = cut(C_1, C_2) \left(\frac{1}{vol(C_1)} + \frac{1}{vol(C_2)} \right)$$

Despite providing more accurate clusters, the crucial issue about *normalized cut* is that finding its minimum is **NP-hard**, and for this reason there exist some efficient approximations that exploit the properties of linear algebra. One of this approximations is based on the **graph Laplacian** or **Laplacian matrix**, which is a matrix defined as:

$$L = D - W$$

, where:

- D is the *diagonal degree matrix*, i.e. $d_{ii} = \deg(i) = \sum_j w_{i,j}$;
- W is the *similarity matrix*, in which the elements of the diagonal are equal to 0 by definition. Moreover, if the graph is unweighted, then W only contains 1s and 0s.

In this sense, the elements of L are given by:

$$L_{i,j} = \begin{cases} \mathrm{d}\,(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

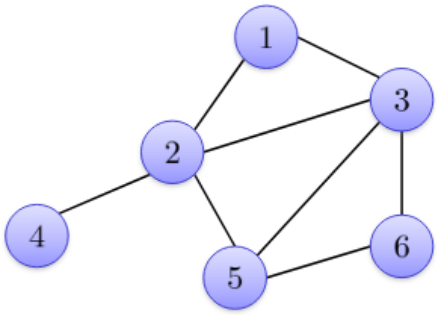
This is an example of the degree matrix D and the affinity matrix W in relation to the graph of the next page.

$$D = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

Figure 107: Degree matrix D.

$$W = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 108: Affinity matrix W.



$$L = \begin{pmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 \\ -1 & -1 & 4 & 0 & -1 & -1 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & -1 & 0 & 3 & -1 \\ 0 & 0 & -1 & 0 & -1 & 2 \end{pmatrix}$$

Assume the weights of edges are 1

Figure 109: Example of laplacian graph.

The *Laplacian matrix* L satisfies the following **properties**:

1. $L \in \mathbb{R}^{n \times n}$ and the sum of rows/columns is always 0;
2. For all vectors x in $\mathbb{R}^n$, we have:

$$x^T L x = \frac{1}{2} \sum_{i,j=1}^{n} w_{ij} \left(x_i - x_j\right)^2 \geq 0$$

This is proved as follows:

$$
\begin{aligned}
x^T L x = x^T (D - W) x = x^T D x - x^T W x = \sum_{i=1}^{n} d_i x_i^2 - \sum_{i,j=1}^{n} x_i x_j w_{ij} \\
&= \frac{1}{2} \left(\sum_i \sum_j d_{ij} x_i x_j - 2 \sum_i \sum_j w_{ij} x_i x_j + \sum_i \sum_j d_{ij} x_i x_j \right) \\
&= \frac{1}{2} \left(\sum_i d_{ii} x_i^2 - 2 \sum_i \sum_j w_{ij} x_i x_j + \sum_j d_{jj} x_j^2 \right) \\
&= \frac{1}{2} \left(\sum_i (\sum_j) w_{ij} \right) x_i^2 - 2 \sum_i \sum_j w_{ij} x_i x_j + \sum_j (\sum_i w_{ji}) x_j^2 \\
&= \frac{1}{2} \left(\sum_i \sum_j w_{ij} x_i^2 - 2 \sum_i \sum_j w_{ij} x_i x_j + \sum_i \sum_j w_{ij} x_j^2 \right) \\
&= \frac{1}{2} \left(\sum_i \sum_j w_{ij} (x_i^2 - 2 x_i x_j + x_j^2) \right) \\
&= \frac{1}{2} \left(\sum_i \sum_j w_{ij} (x_i - x_j)^2 \right)
\end{aligned}
$$

However, since both w_{ij} and $(x_i - x_j)^2$ are ≥ 0, then $x^T L x \geq 0$.

3. L is a **symmetric** and **positive semi-definite** matrix. The symmetry of L follows directly from the symmetry of W and D, while the positive semi-definiteness is a direct consequence of the first property, which shows that $x^T L x \geq 0$. Notice that the positive semi-definiteness implies that all the eigenvalues are non negative, which has crucial consequences in optimization;
4. The smallest eigenvalue of L is 0 and the corresponding eigenvector is the constant 1 vector. Moreover, L has n non-negative, real-valued eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$.

More specifically, an important consequence of these properties is that the **multiplicity**, i.e. the number of eigenvectors associated to a specific eigenvalue, of the smallest eigenvalue $\lambda_1 = 0$ is the **number of connected components** $A_1, \ldots, A_k$ of the graph.

The normalized graph Laplacians There are two matrices which are called normalized graph Laplacians in the literature. Both matrices are closely related to each other and are defined as:

$$
\begin{aligned}
L_{\text{sym}} &= D^{-1/2} L D^{-1/2} = I - D^{-1/2} W D^{-1/2} \\
L_{\text{rw}} &= D^{-1} L = I - D^{-1} W
\end{aligned}
$$

We denote the first matrix by L_{sym} as it is a symmetric matrix, and the second one by L_{rw} as it is closely connected to a random walk.

6.3.1 Solving normalized cut

Any cut (A, B) can be represented by a binary indicator vector x:

$$x_i = \begin{cases} +1 \text{ if } i \in A \\ -1 \text{ if } i \in B \end{cases}$$

, and it can be shown that:

$$\min_x \ \text{Ncut}(x) = \min_y \underbrace{\frac{y^T(D-W)y}{y^T D y}}_{\text{Rayleigh quotient}}$$

subject to the constraint that $y'D = \sum_i y_i d_i = 0$, with $y_i \in \{1, -b\}$. Indeed, y is an indicator vector with 1 in the i-th position if the i-th feature point belongs to A, negative constant $(-b)$ otherwise). Again, this problem is still **NP-hard**, so if we **relax** the constraint of y to be a discrete-valued vector and allow it to take on real values, the original problem

$$\min_y \frac{y^T(D-W)y}{y^T D y}$$

will be equivalent to:

$$\min_y y^T(D-W)y \quad \text{s.t.} \quad y^T D y = 1$$

This amounts to solve a *generalized* eigenvalue problem:

$$\underbrace{(D-W)}_{Laplacian} y = \lambda D y$$

2-ways Ncut Finally, we can provide a **solution** of the *normalized cut* problem by exploiting this **algorithm**:

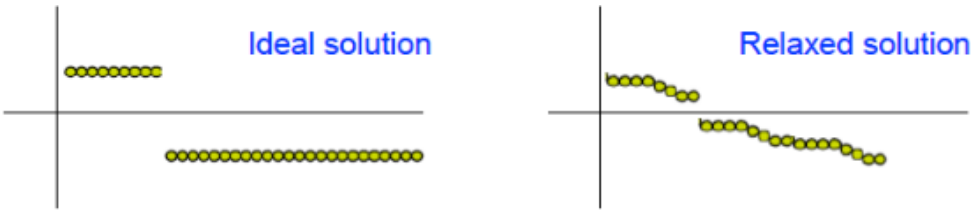
1. Represent the **data points** as a **weighted graph** $G = (V, E)$, compute the weights of each edge and summarize them into D and W;
2. Solve the generalized eigenvalue problem $(D - W)y = \lambda D y$ for the **eigenvector** associated the **second smallest eigenvalue** (we choose the second smallest eigenvalue since the eigenvector associated to the smallest eigenvalue is equal to 1, while the smallest eigenvalue is equal to 0, and it corresponds to the trivial partition $A = V$ and $B = \{\}$);
3. Use the entries of the eigenvector to create a partition of the graph into two parts.

Sometimes there's not a **clear threshold** to split based on the second vector since it takes continuous values. In which way it is possible to choose the splitting point?

- Pick a constant value (0 or 0.5).
- Pick the median value as splitting point.
- Look for the splitting point that has minimum Ncut value:
    1. Choose n possible splitting points.
    2. Compute Ncut value.
    3. Pick minimum.

6.3.2 Relaxation

As we can see, in order to formalize the *minimum cut problem* we had to **relax** the constraint owf y: the goal of relaxation is then to **relax the constraints** of a difficult problem and to solve the simpler problem. If we're lucky, the solution we obtain satisfies the original constraints, so we found a solution of the original problem, otherwise we can choose the nearest point that satisfies the original constraints.
Through relaxation we lose some precision in the final solution. Note that the **original** normalized cut problem returns **binary values** $(-1, 1)$, indicating clustering membership. The **relaxed version**, on the right, returns **continuous values**. It may happen that **some points do not clearly belong to a specific cluster**, as they are close to the margin between the two clusters. For this reason the relaxed solution is **not** always in a **one-to-one correspondence** with the original problem, and choosing a "**correct**" threshold is very important.



6.3.3 Normalized cut with more than 2 clusters

There are two possible approaches if we desire to obtain more than 2 clusters:

Approach #1. It recursively performs the 2-way Ncut algorithm until we obtain the desired number of clusters (not so common).

1. Given a weighted graph $G = (V, E, w)$, summarize the information into matrices W and D.
2. Solve $(D - W)y = \lambda Dy$ for eigenvectors with the smallest eigenvalues.
3. Use the eigenvector with the second smallest eigenvalue to bipartition the graph by finding the splitting point such that Ncut is minimized.
4. Decide if the current partition should be subdivided by checking the stability of the cut, and make sure Ncut is below the prespecified value.
5. Recursively repartition the segmented parts if necessary.

Note: this approach is **computationally wasteful**, only the second eigenvector is used, whereas the next few small eigenvectors also contain useful partitioning information.

Approach #2. Using the first k eigenvectors (far more popular).

1. Construct a similarity graph and compute the unnormalized graph Laplacian L.
2. Compute the k smallest **generalized** eigenvectors $u_1, u_2, \ldots, u_k$ of the generalized eigenproblem $Lu = \lambda Du$.

3. Let $U = [u_1, u_2, \ldots, u_k] \in \mathbb{R}^{n \times k}$, i.e. the columns are the k eigenvectors we computed.
4. Let $y_i \in \mathbb{R}^k$ be the vector corresponding to the ith row of U.

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1k} \\ u_{21} & u_{22} & \cdots & u_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ u_{n1} & u_{n2} & \cdots & u_{nk} \end{bmatrix} = \begin{bmatrix} y_1^T \\ y_2^T \\ \vdots \\ y_n^T \end{bmatrix}$$

5. Thinking of y_i's as points in $\mathbb{R}^k$, cluster them with k-means algorithms. Why do we use k-means? The rows of U allow to map the vertices of the graph in a 2-dimensional vector (like a projection), and it was proved that this mapping facilitates the use of the k-means algorithm.

Note: the number k of clusters must be known in advance.

6.3.4 Spectral clustering vs k-means

First of all, let us define the intuition behind **spectral clustering**: its goal is to cluster data that is connected but not necessarily compact or clustered within convex boundaries. This algorithm is very similar to the previous one, but it has 2 main differences; it works as follows

1. Construct a similarity graph and compute the normalized graph Laplacian L_{sym}.
2. Embed data points in a low-dimensional space (spectral embedding), in which the clusters are more obvious, computing the k smallest eigenvectors $v_1, \ldots, v_k$ of L_{sym}.
3. Let $V = [v_1, \ldots, v_k] \in \mathbb{R}^{n \times k}$.
4. Form the matrix $U \in \mathbb{R}^{n \times k}$ from V by normalizing the row sums to have norm 1, that is:

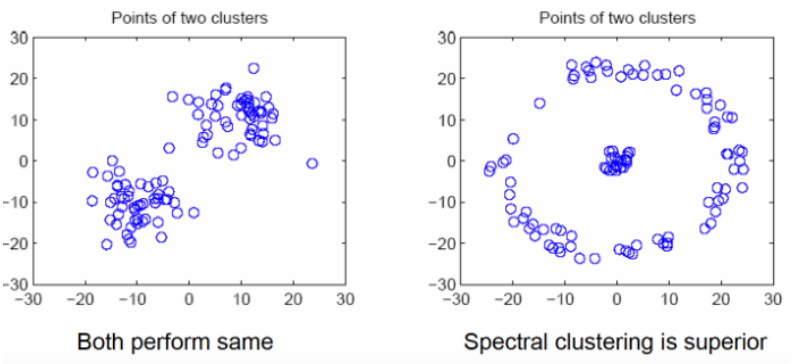
$$u_{ij} = \frac{v_{ij}}{\left(\sum_k v_{ik}^2\right)^{1/2}}$$

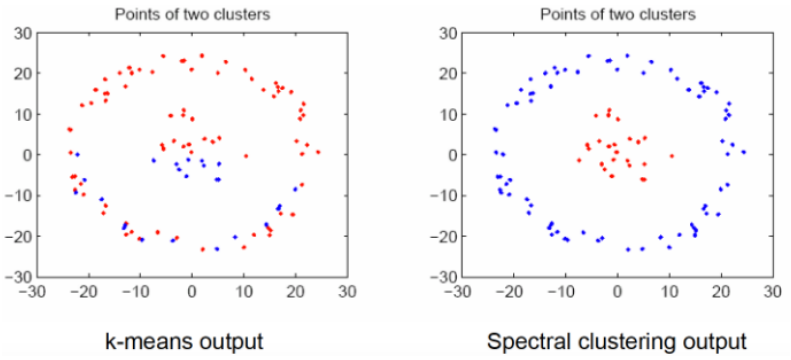
5. For $i = 1, \ldots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the ith row of U.
6. Cluster the points y_i with $i = 1, \ldots, n$ with the k-means algorithm into clusters $C_1, \ldots, C_k$.

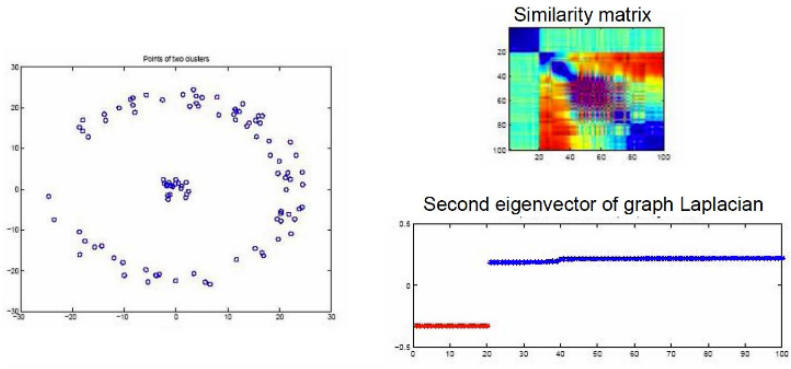
The differences with the previous algorithm are following:

- We compute a **normalized graph Laplacian** L_{sym};
- The rows of U are normalized to norm 1.

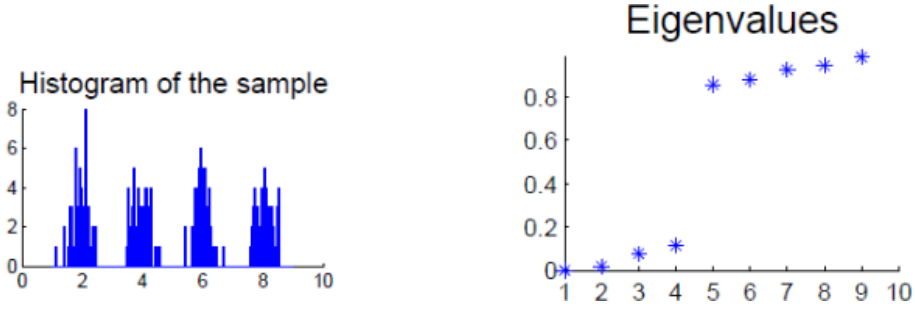
Applying k-means to Laplacian eigenvectors allows us to find **cluster** with **non-convex boundaries**.





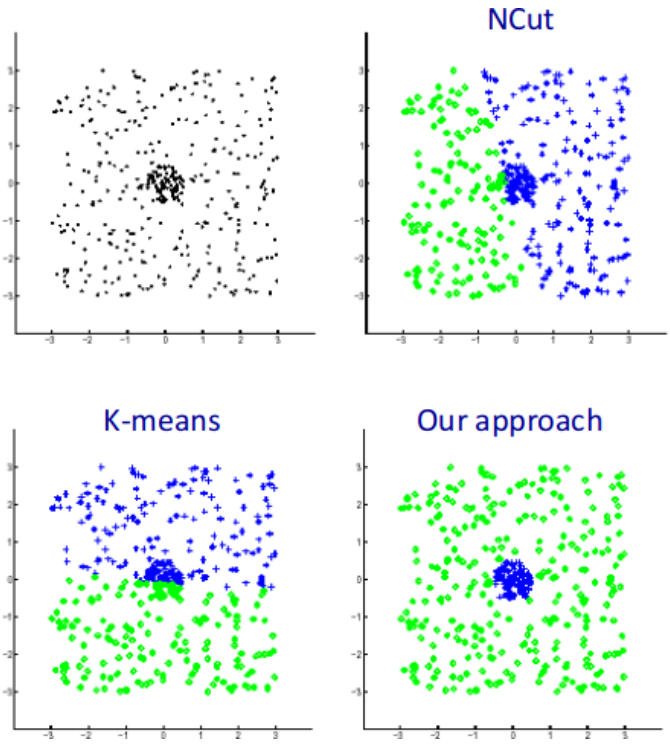


As we said before, one of the main **issue** of k-means is represented by the choice of the value of k: one of the possible solutions (called **eigengap heuristic**) is to choose k such that all eigenvalues $\lambda_1, \ldots, \lambda_k$ are very small, but λ_{k+1} is relatively large. In this way, the choosing of k maximizes the eigengap (difference between consecutive eigenvalues) $\delta_k = |\lambda_k - \lambda_{k-1}|$.



7 Dominant-set clustering

A very important limitation of the clustering algorithms we've seen so far (especially the **partition-based** one) is that they're not able to separate the structure of the data from the clutter: this happens because they're goal is to locate distinct coherent clusters, but they do not work very well in presence of noise.



Indeed, in certain real-world problems, natural groupings are found among only on a small subset of the data, while the rest of the data shows little or no clustering tendencies. In such situations, it is often **more important** to **cluster a small subset** of the data very well, **rather** than **optimizing** a **clustering criterion** over **all the data points**, particularly in application scenarios where a large amount of noisy data is encountered. Moreover, while partitional approaches impose that each element cannot belong to more than one cluster, this new approach allows to also consider nodes belonging to two different clusters, hence considering the hypothesis of **overlapping clusters**.

7.1 Graph-theoretic definition of a cluster

Data to be clustered are represented as an undirected weighted graph with no self-loops: $G = (V, E, \omega)$, where $V = \{1, \ldots, n\}$ is the vertex set, $E \subseteq V \times V$ is the edges set and $w : E \to \mathbb{R}_+^*$ is the positive weight function. Vertices represent data points, edges neighborhood relationships and edge-weights similarity relations. G is then represented with an adjacency matrix A, such that $a_{ij} = \omega(i, j)$. Since there are not self-loops we have that $\omega(i, i) = 0$ (main diagonal equal to 0). From now on, if not otherwise stated, A will represent such matrix.
There is **not** an **unique** and well defined **definition** of **cluster**, but the available literature agrees in two conditions that a cluster should satisfy:

- **High internal homogeneity**, also named *internal criterion*. It means that all the objects inside a cluster should be highly similar to (or have low distance from) each other;
- **High external inhomogeneity**, also named *external criterion*. It means that objects coming from different clusters have low similarity (or high distance).

The idea of these criterion is that **clusters** are groups of objects which are **strongly similar** to each other if they belong to the same cluster, otherwise they are **highly dissimilar**.

Basic definitions. Let $S \subseteq V$ be a nonempty subset of vertices and $i \in S$. The **average weighted degree** of i w.r.t. S is defined as:

$$\text{awdeg}_S(i) = \frac{1}{|S|} \sum_{j \in S} a_{ij} \tag{1}$$

, i.e. it represents the **average similarity** between entity i and the rest of the entities in S. It can be observed that $\text{awdeg}_{\{i\}}(i) = 0 \ \forall i \in V$, since we have no self-loops.

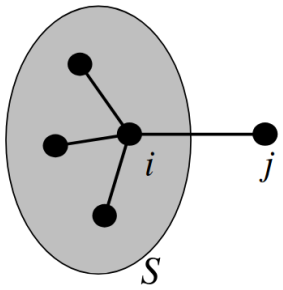


Figure 110: Average weighted degree.

We now introduce a new quantity ϕ such that if $j \notin S$:

$$\phi_S(i, j) = a_{ij} - \text{awdeg}_S(i) \tag{2}$$

Intuitively, $\phi_S(i, j)$ measures the **relative similarity** between i and j with respect to the **average similarity** between i and its neighbors in S. This measures can be either positive or negative.

Let $S \subseteq V$ be a nonempty subset of vertices and $i \in S$. The **weight** of i w.r.t. S is:

$$w_S(i) = \begin{cases} 1 & \text{if } |S| = 1 \text{ (singleton)} \\ \sum_{j \in S \setminus \{i\}} \phi_{S \setminus \{i\}}(j, i) w_{S \setminus \{i\}}(j) & \text{otherwise} \end{cases} \quad (3)$$

Furthermore, the **total weight** of S is defined to be $W(S) = \sum_{i \in S} w_S(i)$.

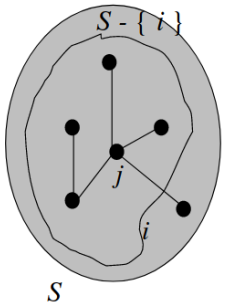


Figure 111: Weight of i w.r.t. the elements in S.

Note that $w_{\{i,j\}}(i) = w_{\{i,j\}}(j) = a_{ij}\ \forall i, j \in V \wedge i \neq j$. Then, $w_S(i)$ is computed simply as a function of the weights on the edges of the sub-graph induced by S.
Intuitively, $w_S(i)$ gives a measure of the **similarity** between i and $S \setminus \{i\}$ with respect to the **overall similarity** among the vertices of $S \setminus \{i\}$. In other words, it represents **how similar (important)** i is **with respect to the entities** in S. An important property of this definition is that it induces a sort of **natural ranking** among the **vertices** of the graph.

Figure 112: Examples of total weight

As we can see, in the first example we should not add node 1 to the cluster {1,2,3}, since it has a low similarity compared with the other nodes (as it can be seen from the value of the total weight), while in the second case we would add node 1 in order to obtain a larger and more coherent cluster.

Dominant set. A nonempty subset of vertices $S \subset V$ such that $W(T) > 0$ for any nonempty $T \subseteq S$, is said to be a **dominant set** if:

- $w_S(i) > 0,\ \forall i \in S$ (*internal homogeneity*)
- $w_{S \cup \{i\}}(i) < 0,\ \forall i \notin S$ (*external homogeneity*)

These conditions correspond to cluster properties (**internal homogeneity** and **external in-homogeneity**). Informally we can say that the **first condition** requires that **all the nodes** in the clusters are **important** for it. The **second** one assumes that if we consider a **new point** in the cluster, the **cluster cohesiveness will be lower**, meaning that the current cluster should be already maximal.
By definition, dominant sets are expected to capture **compact structure**s. Moreover, this definition is equivalent to the one of maximal clique problem when applied to unweighted graphs.

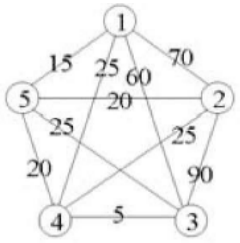


Figure 113: The set of vertices {1,2,3} is dominant.

7.2 Connections of dominant sets

Dominant sets have intriguing connections with:

- **Game theory**, and concepts like Nash equilibria;
- **Optimization theory**, in particular they are local maximizers of (continuous) quadratic problems;
- **Graph theory** and **maximal cliques**;
- **Dynamical systems theory**.

7.2.1 Game theory

Game theory is the study of **mathematical models** of strategic interaction between rational decision-makers. In this sense we can model a new **clustering game** with the following properties:

- **Symmetric game**, the payoffs for playing a particular strategy depend only on the other strategies employed, not on who is playing them;
- **Complete knowledge**, payoffs, strategies and types of players are known;
- Pre-existing set of **pure strategies**. Players do not behave "rationally" but act according to a pre-programmed behavioral pattern (pure strategy).

Data points V are the pure strategies available to the players and the similarity matrix A represents the **payoff matrix**, which resumes the revenues that each player obtains when a pair of strategies is played. The values A_{ij} and A_{ji} are the revenues obtained by player 1 and player 2 considering that they have player strategies $(i, j) \in V \times V$. A **mixed strategy** $x = (x_1, \ldots, x_n)^T \in \Delta$ is a probability distribution over the set of pure strategies, which models a stochastic playing strategy of a player. If player 1 and

2 play mixed strategies $(x_1, x_2) \in \Delta \times \Delta$, then the expected payoffs for the players are: $\mathbf{x_1^T A x_2}$ and $\mathbf{x_2^T A x_1}$ respectively. The goal of the two players of course is to maximize their resulting revenue as much as possible. During the game each player extracts an object (i, j) and the resulting revenue is associated according to the payoff matrix A. Since we are considering A as equal to the similarity matrix, we can say that in order to **maximize** their revenues the two players should **coordinate** their strategies so that the **extracted objects belong to the same cluster**. In other words, only by selecting objects belonging to the same cluster, each player is able to maximize his expected payoff. The unique difference is that the similarity between two equal entities is 0, meaning that $A_{ii} = 0$. The desired condition is that the two players reach a **symmetric Nash equilibrium**, a state in which the two players agree about the cluster membership. A **Nash equilibrium** is a mixed-strategy profile $(x_1, x_2) \in \Delta \times \Delta$ such that no player can improve the expected payoff by changing his playing strategy, given the opponent's strategy being fixed. In other words, it is a **configuration of strategies** for which **no player** will deviate from it for its convenience since there is **no incentive** to change choice. This concept can be expressed with the following expression:

$$y_1^T A x_2 \leq x_1^T A x_2 \qquad y_2^T A x_1 \leq x_2^T A x_1 \qquad \forall (y_1, y_2) \in (V \times V).$$

A Nash equilibrium is **symmetric** if $x_1 = x_2$, meaning that considering a symmetric Nash equilibrium $x \in \Delta$ the two conditions hold in a unique one:

$$y^T A x \leq x^T A x$$

This condition satisfies the **internal homogeneity** criterion required by the dominant set definition, but it does **not** include any kind of constraint that guarantees the **maximality condition**. In order to satisfy this condition it is necessary to look for a different type of Nash Equilibrium, known as **Evolutionary Stable Strategy (ESS)**.

ESS. A symmetric Nash equilibrium $x \in \Delta$ is an **evolutionary stable strategy** (ESS) if it satisfies also:

$$y^T A x = x^T A x \implies x^T A y > y^T A y \qquad \forall y \in \Delta \setminus \{x\}$$

$$y^T A x = x^T A x \implies x^T A y < x^T A x \qquad \forall y \in \Delta \setminus \{x\}$$

Even if the strategy y provides the same payoff of the strategy x, it is better to play x since the payoff against itself is greater than the one provided by y. The two strategies x and y represents two Nash Equilibrium, but only x is an ESS.
In conclusion we can say that the **ESSs of the clustering game** with affinity matrix A are in **correspondence** with **dominant sets** of the same clustering problem instance. However, we can also conclude that **ESSs** are in **one-to-one** correspondence to **(strict) local solutions of StQPs** (Standard Quadratic optimization Problems).
It is possible to say that ESSs abstract well the main characteristic of a cluster:

- **Internal coherency:** High mutual support of all elements within the group.
- **External incoherency:** Low support from elements of the group to elements outside the group.

7.2.2 Optimization theory

Clusters are commonly represented as n-dimensional **vectors** expressing the participation of each node to a cluster. Large numbers denote a strong participation, while zero values no participation. The **cohesiveness** of a cluster can be computed using:

$$f(x) = x^\top Ax \tag{1}$$

where A is the **symmetric** real-valued matrix with null diagonal. **Clustering** can now be formulated as the problem of **finding the vector** x that **maximizes** f. The objective function has to be **normalized**. For this aim simplex constraints are imposed. This yields the following **standard quadratic optimization problem** whose **local solution** corresponds to a **maximally cohesive cluster**:

$$\begin{array}{ll} \max & x^T Ax \\ \text{s.t.} & x \in \Delta \end{array} \tag{2}$$

where

$$\Delta = \{x \in \mathbb{R}^n : x \geq 0 \wedge x^\top x = 1\} \tag{3}$$

is the **standard simplex** of $\mathbb{R}^n$.

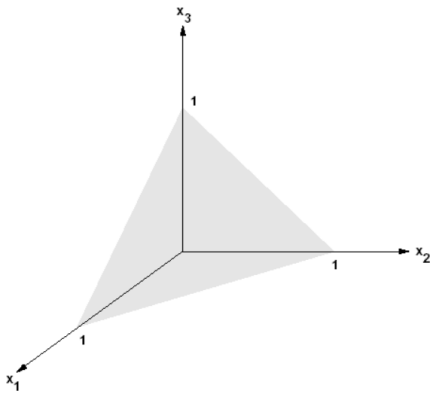


Figure 114: Standard simplex on $\mathbb{R}^3$.

As we can see, this problem is quite similar to the one presented in the previous Chapter, but the difference relies in the constraint.
In conclusion **dominant sets** can be put in **one-to-one correspondence** (modulo a technical condition) with **strict local maximizers** of a **quadratic function over the simplex**. As opposed to many other clustering algorithms, which try to find the global optimum of some energy function, dominant sets can be found by mining local solutions.

7.2.3 Graph theory and maximal cliques

Suppose we have a **binary similarity matrix** and an unweighted undirected graph $G = (V, E)$, then:

- A **clique** is a subset of mutually adjacent vertices;

- A **maximal clique** is a clique that is not contained in a larger one;

It was proved that **EES's** are in **one-to-one correspondence** to **maximal cliques** of G.

7.3 Finding dominant sets

One of the major **advantages** of using **dominant sets** is that the procedure that allows to find them can be written with few lines of code. There are several dominant set clustering approaches:

- To get a **single** dominant set cluster use **replicator dynamics**;
- To get a **partition** use a simple *peel-off* strategy: iteratively find a dominant set and remove it from the graph, until all vertices have been clustered;
- To get **overlapping clusters**, enumerate dominant sets.

The **replicator dynamics** (RD) are **deterministic game dynamics** that have been developed in evolutionary game theory. They consider an idealized scenario whereby individuals are repeatedly drawn at random from a large, ideally infinite, population to play a two-player game. In contrast to classical game theory, here players are not supposed to behave rationally or to have complete knowledge of the details of the game. They act instead according to an inherited behavioral pattern, or pure strategy, and it is supposed that some evolutionary selection process operates over time on the distribution of behaviors.
Let $x_i(t)$ be the population share playing pure strategy i at time t. The state of the population at time t is: $x(t) = (x_1(t), \ldots, x_n(t)) \in \Delta$.
We define an evolution equation, derived from Darwin's principle of nature selection:

$$\dot{x_i} = x_i \; g_i(x)$$

where g_i specifies the rate at which pure strategy i replicates.

$$\frac{\dot{x_i}}{x_i} \propto \text{payoff of pure strategy } i \text{ - average population payoff}$$

which yields:

$$\dot{x_i} = x_i[(Ax)_i - x^T Ax]$$

where $(Ax)_i$ is the i-th component of the vector and $x^T Ax$ is the average payoff for the population. If we have a result which is better than the average strategy, then there is an improvement. We can see $\dot{x_i}$ as the fraction of the total number of players that's using strategy i. As $\dot{x_i}$ increases, so does the number of players using it; conversely, if $\dot{x_i}$ decreases, the strategy will likely disappear.

Theorem. A point $x \in \Delta$ is a Nash equilibrium if and only if x is the limit point of a replicator dynamics trajectory starting from the interior of Δ. Furthermore, if $x \in \Delta$ is an ESS, then it is an asymptotically stable equilibrium point for the replicator dynamics.

Assuming that the payoff matrix A is symmetric ($A = A^T$), we call this type of game as a doubly symmetric game. Thanks to this assumption we can derive some conclusions:

- (***Fundamental Theorem of Natural Selection***.) For any doubly symmetric game, the average population payoff $f(x) = x^T Ax$ is strictly increasing along any non-constant trajectory of replicator dynamics, meaning that $\frac{df(x(t))}{dt} \geq 0 \ \forall t \geq 0$, with equality if and only if $x(t)$ is a stationary point.
- (***Characterization of ESSs***.) For any doubly symmetric game with payoff matrix A, the following statements are equivalent:
  - $x \in \Delta^{ESS}$
  - $x \in \Delta$ is a strict local maximizer of $f(x) = x^T Ax$ over the standard simplex Δ.
  - $x \in \Delta$ is asymptotically stable in the replicator dynamics.

A well-known **discretization** of replicator dynamics, which assumes non-overlapping generations, is the following (assuming a non-negative A):

$$x_i(t+1) = x_i(t)\frac{A(x(t))_i}{x(t)^T Ax(t)}$$

which inherits most of the dynamical properties of its continuous-time counterpart. The idea here is the same as above: we are evolving dominant strategies. Since $A(x(t))_i$ is the payoff for strategy i and $x(t)^\top Ax(t)$ is the average payoff, if the ratio is greater than 1 then the strategy is dominant and the derivative is positive increasing.

```
distance=inf;
while distance>epsilon
    old_x=x;
    x = x.*(A*x);
    x = x./sum(x);
    distance=pdist([x,old_x]');
end
```

Figure 115: MATLAB implementation of discrete-time replicator dynamics

The components of the **converged vector** give us a measure of the **participation** of the corresponding vertices in the cluster, while the **value** of the **objective function** measures the **cohesiveness** of the cluster.

7.4 Image segmentation

An image is represented as an edge-weighted undirected graph, where **vertices** correspond to **individuals pixels** and edge-weights reflect the **similarity** between pairs of vertices. The **clustering** problem consists on extracting **dominant sets** from an input image, below is proposed the pseudo-code version of the solution with dominant sets:

```
Partition_into_dominant_sets(G)
Repeat
   find a dominant set
   remove it from graph
until all vertices have been clustered
```

Figure 116: Pseudo-code for image segmentation.

Remember that to find a single dominant set we used replicator dynamics.

7.5 Properties

- **Well separation between structure and noise.** In such situations it is often more important to cluster a small subset of the data very well, rather than optimizing a clustering criterion over all the data points, particularly in application scenarios where a large amount of noisy data is encountered;
- **Overlapping clustering**. In some cases we can have that two distinct clusters share some points, but partitional approaches impose that each element cannot be long to more than one cluster;
- Dominant sets can be found by mining **local solutions**, so it is not necessary to look for global solutions;
- Performs very well in presence of **noise** and **outliers**;
- Makes **no assumptions** on the **structure** of the **affinity matrix**, being it able to work with asymmetric and even negative similarity functions;
- Does **not** require a **priori knowledge** on the **number of clusters** (since it extracts them sequentially).
- Leaves **clutter** elements **unassigned** (useful, e.g., in figure/ground separation or one-class clustering problems);
- Generalizes naturally to **hypergraph clustering problems**;
- It allows to rank cluster's elements according to their **centrality**.