Image and Video Understanding

Academic Year 2022/2023

Nicola Aggio 880008

Index

1 Introduction

In introducing the analysis of image, videos and computer vision, we first ask ourselves **"What does it mean to see?"**. An interesting answer was provided by **David Marr (1982)**, one of the father of visual perception, *"The plain man's answer would be, to know **what** is **where** by looking"*. In this sense, his idea was to associate the concept of seeing to the task of **object detection**, as represented in the following picture:

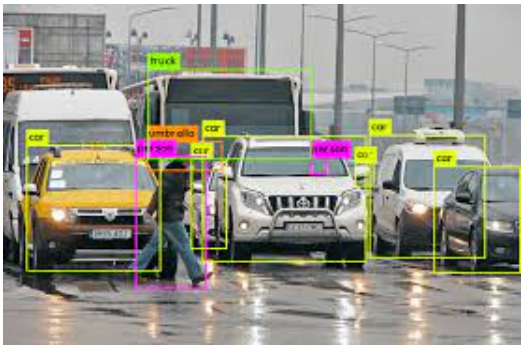


Figure 1: Object detection problem

1.1 Theory of vision: from Pythagoras to Marr

The very first theories about vision were:

- the **emission theory**, which thought that the eye was capable of visualizing the objects in the world by using a "fire". An important feature of this theory is that it considered visual perception from a mathematical point of view, and for this reason it was supported by Pythagoras, Euclid and Empedocles;
- the **intromission theory**, which thought that the images came from the objects and entered our eyes. This theory was supported by Democritus, Epicurus and Lucretius;
- **Plato's view**, which can be considered a compromise between the previous theories, in the sense that he believed that the image was captured by the eye using both the "fire" of the eye itself and the "body" of vision of the object;
- **Alhazen's theory**, which was introduced in its *Book of Optics*, representing a new theory of perception and pointing out the weakness of the previous theories;
- **Kepler's modern theory** of retinal images, in which he understood how the images are formed in our eyes.

Moreover, along with the different theories we introduced, two fundamental intellectual traditions dominated the biological field of vision:

- the **nativism** (Kant, Cartesio etc..), which thought that there's an infrastructure that allows humans to visualize the objects;

- the **empiricism** (Locke, Hume etc..), which on the other hand thought that humans perceived the external world by learning.

Among the Empiricists, a very interesting point of view was provided by **Helmotz**, who believed in the *vision as an unconscious inference*. More specifically, he thought that visual perception is:

- *unconscious*, because we cannot explain why we see or why we recognize things, so we cannot provide a solution to the vision problem since we are not aware of what that is;
- characterized by an *inference process*, i.e. that our brain infers the image we see by some other basic features. An example of this inference process is provided by Picture 1.1: in this case we infer the central white triangle from the shapes and the colours of the surrounding objects, even if the triangle is not actually represented.

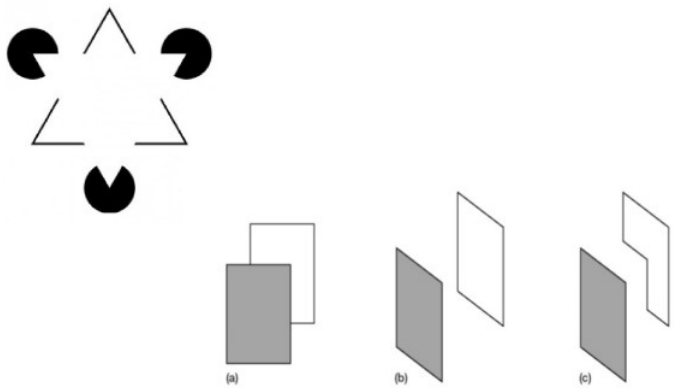


Figure 2: Example of an inference processes

Another biological example of this inference process is provided by our eye: we know that only the center of the retina (fovea) is composed of perceptors that allow to capture the colours of an object, but the brain collects these samples and forms an image which entirely colored.

Another important contribution was brought by the **Gestalt school**, whose goal was to establish some new laws concerning the visual perception. Despite failing in this project, they still managed to introduce some important notions, for example the one of *grouping*, which can be considered nowadays as the *clustering problem*, one of the main issues in Computer Vision.
Finally, **David Marr** in 1982 provided a new theory about vision, which was based on a mathematical perspective, neuroscience and IT. More specifically, he introduced three levels of understanding the problem of vision :

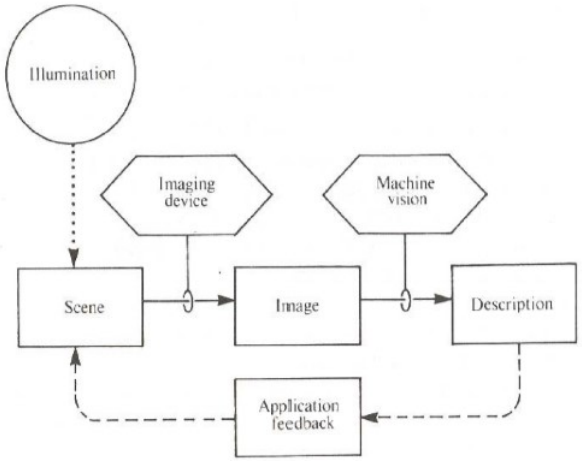
- the **computational level**, i.e. what does the system do? What problems does it solve or overcome?
- the **algorithmic** or **representational level**, i.e. how does the system do what it does? What representations does it use and what processes does it employ to build and manipulate the representation?

- the **implementation** or **physical level**, i.e. how is the system physically realized? This level addresses the actual implementation of the algorithms.

However, we can see that Marr did not consider learning in his theory, and this was pointed out by **Tomaso Poggio**, who at the contrary believed in learning as the top level of understanding and as the tool which is necessary *"to build intelligent machines that are able to learn to see (and think) without the need to be programmated to do it"*.

1.2 Computer Vision

We can describe the goal of Computer Vision as the one of producing a symbolic description of what is being imaged, and for this reason it can be viewed as an inversion of the image process.



Actually, Computer Vision can be considered as an intersection of many other **disciplines**, such as:

- *Image processing*, which takes as input an image and produces another (possibly) better image. An example of image processing algorithm is image compression;
- *Pattern recognition*, which classifies object and detects patterns on them;

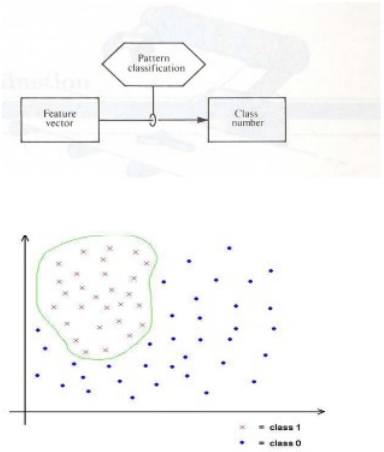


Figure 4: Pattern recognition

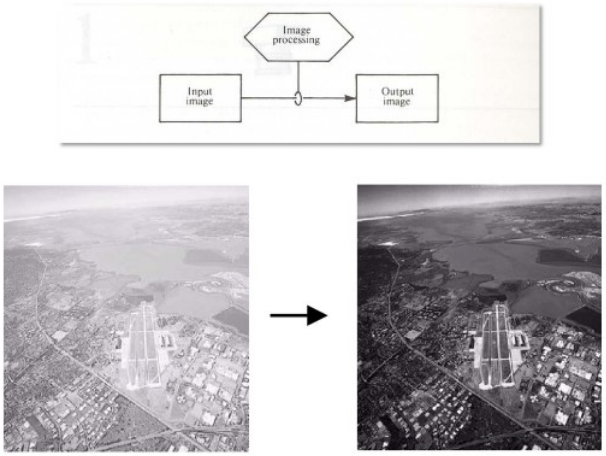


Figure 3: Image processing

- *Scene analysis*, which provides a high-level symbolic description of the scene;

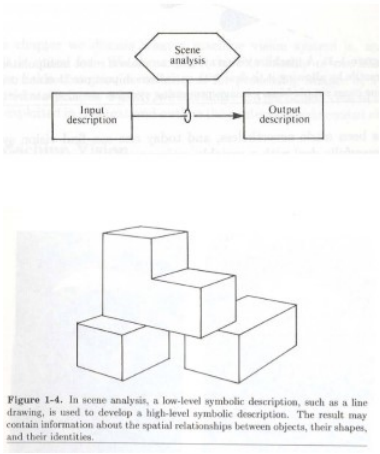


Figure 5: Scene analysis

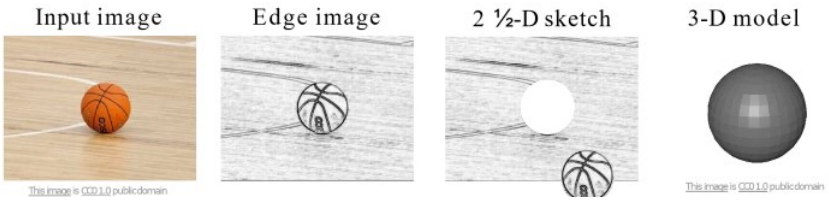
- *Graphics*, which given a model produces an image (synthesis problem).

In general, the historical landmarks in the field of Computer Vision were:

- in 1963, Larry Roberts introduced some algorithms, for example edge detection, focusing on the block worlds, i.e. a symplified world in which the objects are represented as blocks;
- the *Summer Vision Project* in the summer of 1966, in which the students were asked to produce a visual system for pattern recognition. The task of this project underlines the overall underestimation of the problem of CV;
- in the 70's, Marr's idea was to build models and representations of the input image which were more and more abstract. In this sense, the representations were:

  1. *Primal sketch*, which is a very primitive representation;

2. *2-$\frac{1}{2}D$*, which captured the local surface orientation and the discontinuities in depth and in surface orientation;
3. *3-D model representation.*



- in the 70's, other 3D representations were introduced, such as the *Generalized Cylinder* or the *Pictorial Structure*, whose goal was to decompose real-world objects and their relations and representing them as graphs;
- in 1987 and 1999, Lowe introduced the task of *feature extraction* (SIFT), i.e. he thought that vision can be described as the process of extracting complex features from an image;
- the problem of *Image segmentation and clustering*, i.e. partition the image into segments that are coherent from a colour point of view. This problem was the starting point for many other algorithms;
- *Face detection*, which led to the first successful algorithm (Viola Jones, 2001), which was both correct and fast and it was implemented for many years;
- *Histogram of Gradients (HoG), 2005*
- *Deformable Part Model, 2009*

Then, people started building datasets for training Machine Learning algorithms for many CV problems, which pushed the community both to develop more accurate ML algorithms and to have an objective measure of the performances of the CV algorithms. An example of this tendency was the *PASCAL Visual Object Challenge (2006-2012)*, which was an object detection challenge, or the *ImageNET (2009)*, a very large dataset containing various labeled images with a large number of categories (22K categories for 14M images). In 2009, the *Large Scale Visual Recognition Challenge* was instituted, and year after year the performances of the ML algorithms increased, until 2012, when the first Deep Learning approach was introduced, representing a huge change in CV history.
The idea behind the Deep Learning approach was to get rid of the standard approach, consisting in designing good features and then feed the ML algorithm, and to exploits the learn phase of the algorithms to learn a feature hierarchy. In this sense, each layer of the deep Neural Network extracts features from the output of the previous one, creating a hierarchy and producing a final classifier. Despite being an old idea, Dee Learning was introduced only in 2012 because of the large availability of data and computing power (GPUs).

2 Machine Learning Basics

2.1 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 6: 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.

2.1.1 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.

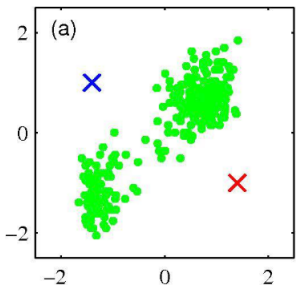


Figure 7: Initialization with $K = 2$.

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

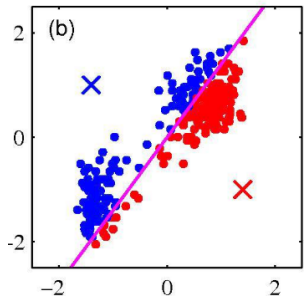


Figure 8: Iterative step 1.

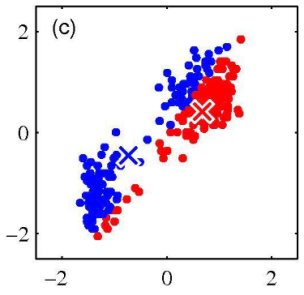


Figure 9: Iterative step 2.

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

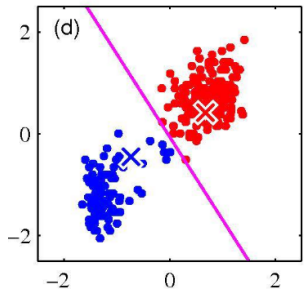


Figure 10: Repeat until convergence.

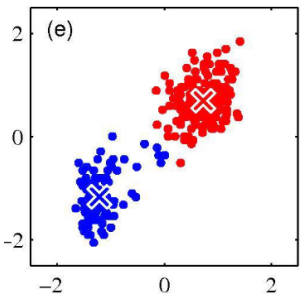


Figure 11: Final output.

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.

2.2 Classification

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.

2.2.1 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.

2.2.2 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..

2.2.3 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.

2.3 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).

2.3.1 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.

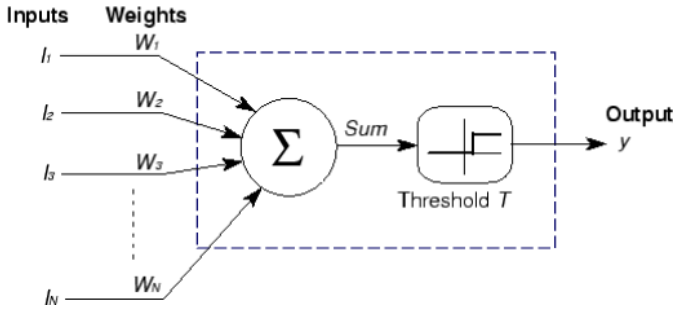


Figure 12: McCulloch and Pitts neuron representation.

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.

2.3.2 Properties

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

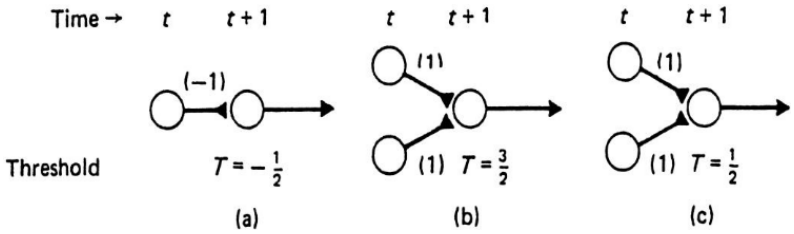


Figure 13: Three elementary logical operations (a) **negation**, (b) **and**, (c) **or**. In each diagram the states of the neurons on the left are at time t and those on the right at time $t+1$.

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

2.3.3 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.

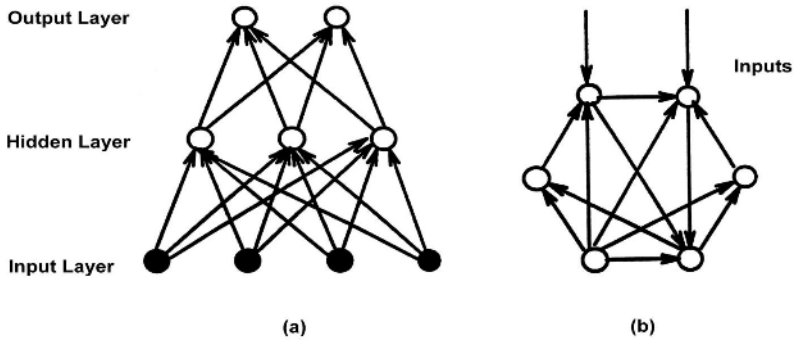


Figure 14: *(a)* feedforward network - *(b)* feedback network

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.

2.3.4 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.

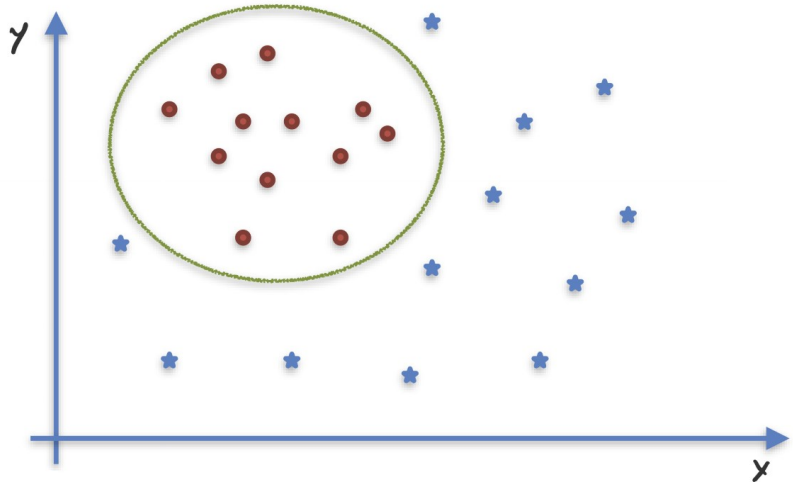


Figure 15: Classification problem

2.3.5 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:

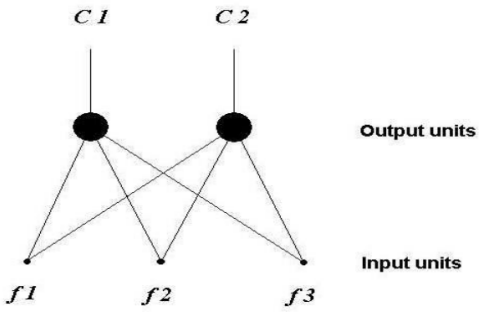


Figure 16: Example of NN with 3 features input and 2 class labels as output.

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

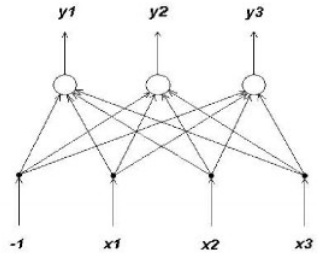


Figure 17: We can remove thresholds by adding an input neuron clamped at -1

2.3.6 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.

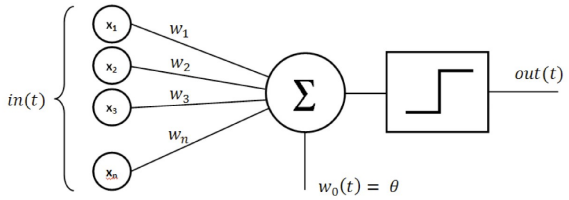


Figure 18: Representation of a perceptron

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:

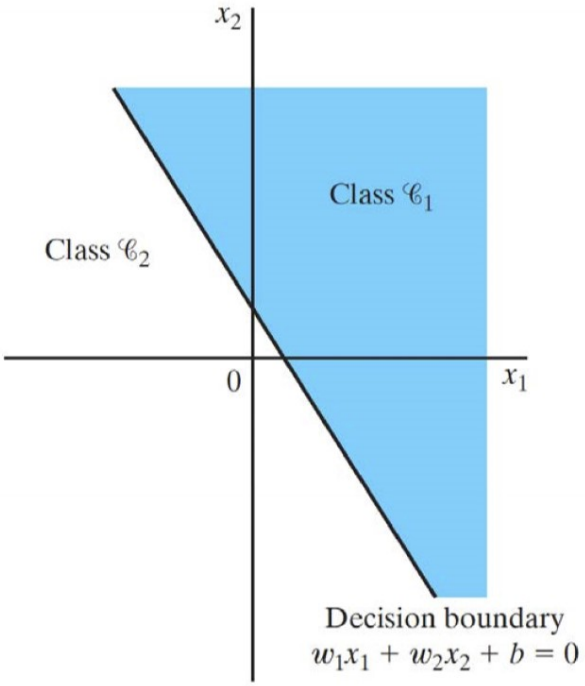


Figure 19: Geometry interpretation of the perceptron

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 $w_1x_1 + w_2x_2 < 0$

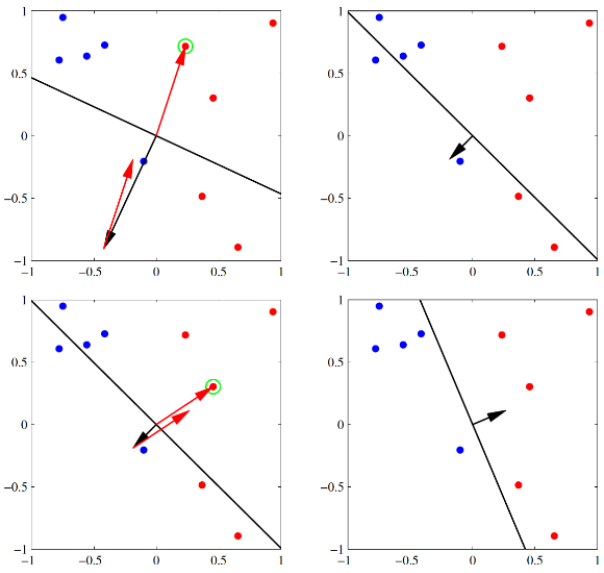


Figure 20: Perceptron learning algorithm procedure.

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.

2.3.7 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.

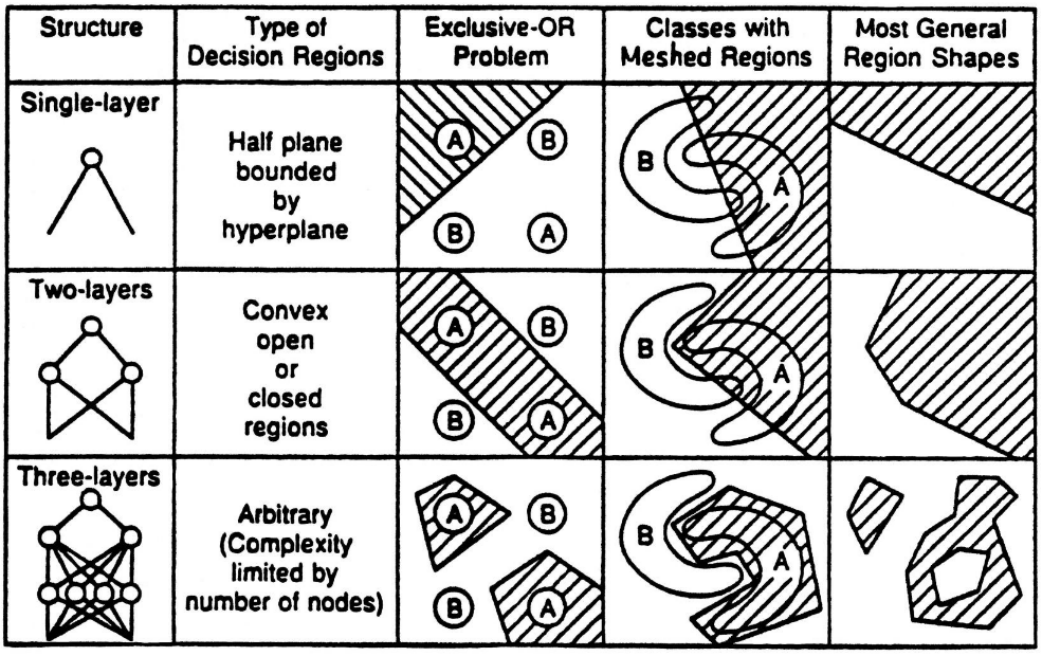


Figure 21: A view of the Role of Units

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.

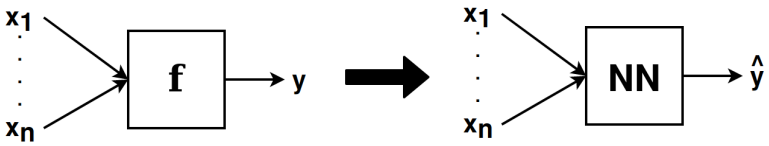


Figure 22: Neural Network function model.

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.

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

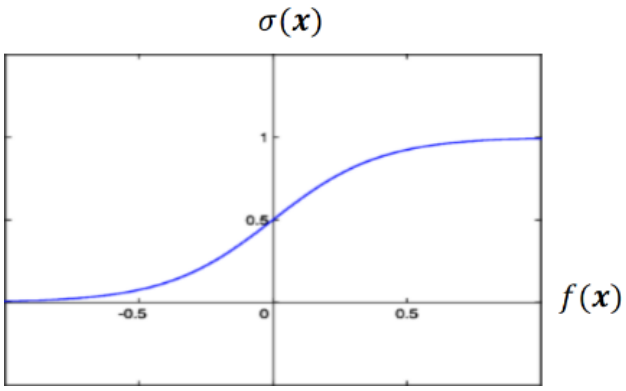


Figure 23: Sigmoid function.

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.

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

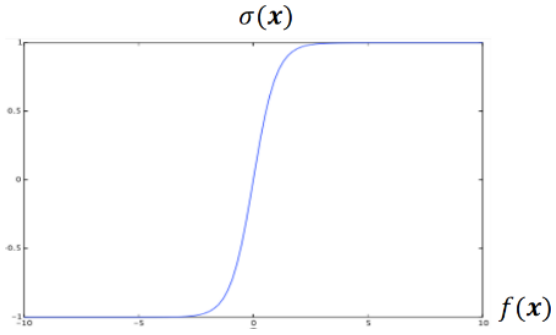


Figure 24: Hyperbolic tangent function.

2.3.8 The 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.

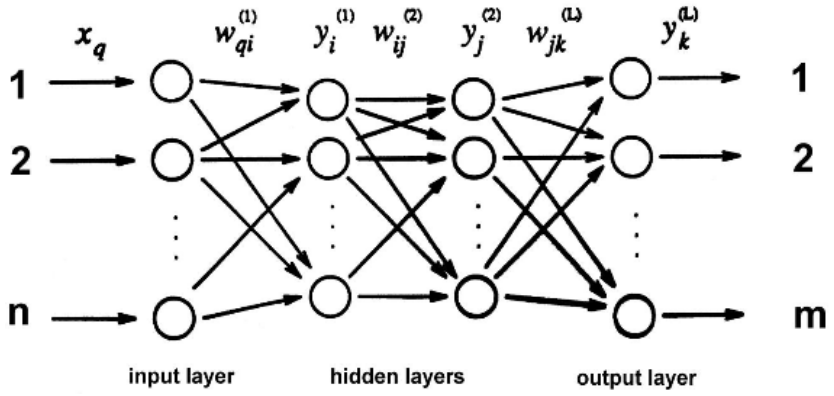


Figure 25: Back-propagation schema. W_{ij}^{l} represents the weight on connection between the i^{th} unit in layer $(l-1)$ to j^{th} unit in layer l.

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:

$x^{\mu}(\mu = 1, \ldots, N)$ is the network **input** 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;

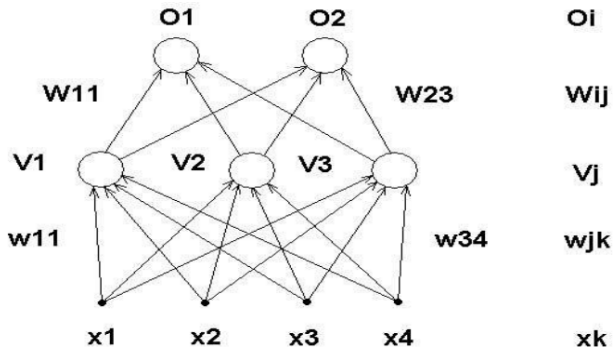
Now we need to define a method for computing the partial derivatives of the loss function efficiently.
To compute the partial derivatives we use the **error back propagation** algorithm, which consists of two stages:

- **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.

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

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

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

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

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

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

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.

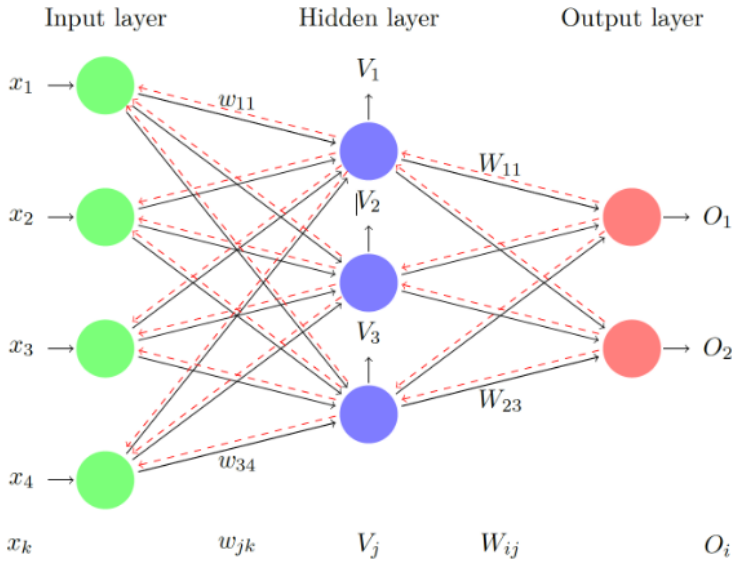


Figure 26: Error Back-propagation representation.

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

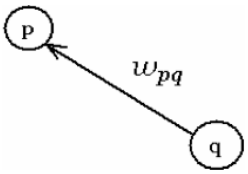


Figure 27: Locality of back-propagation

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 $0.02, 0.0476, 0.049, 0.0505$.

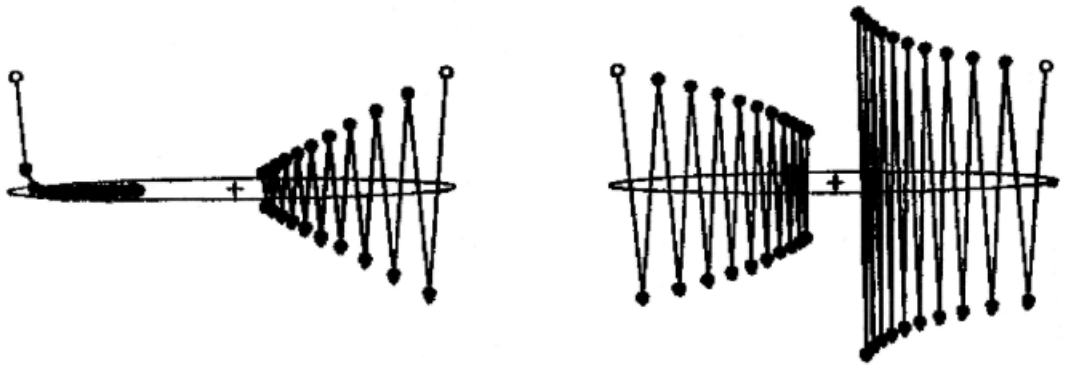


Figure 28: The Role of the Learning Rate. The minimum is at the + and the ellipse shows a constant error contour.

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.

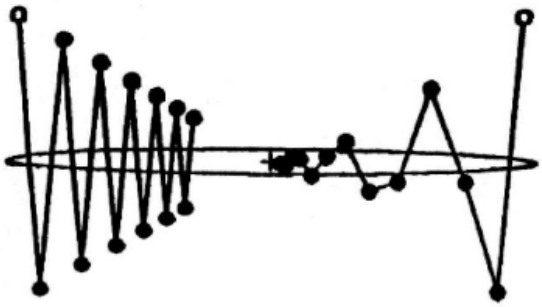


Figure 29: Momentum term application.

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.

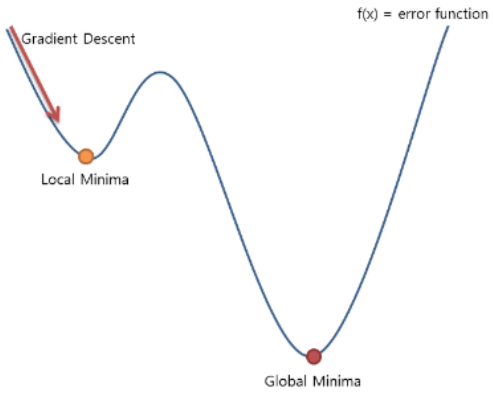


Figure 30: The problem of local minima.

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.

2.3.9 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.

2.4 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.

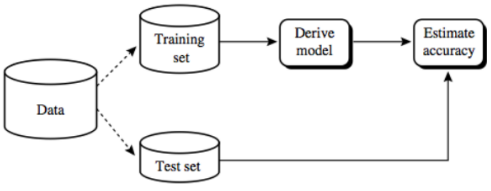


Figure 31: Training set vs Test set.

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.

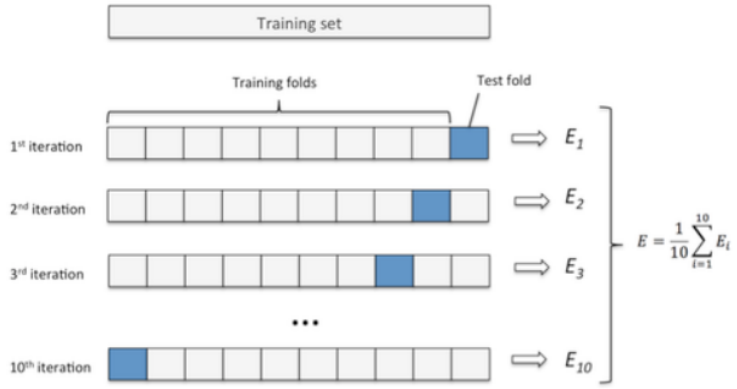


Figure 32: Cross validation example using **leave-one-out** technique, i.e. the size of the test fold is 1.

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**.

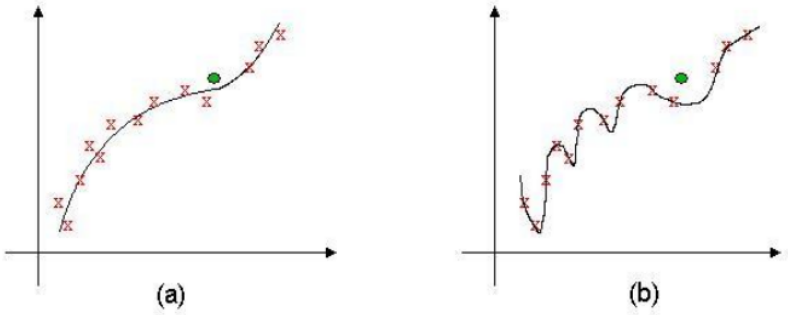


Figure 33: Example of overfitting.

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.

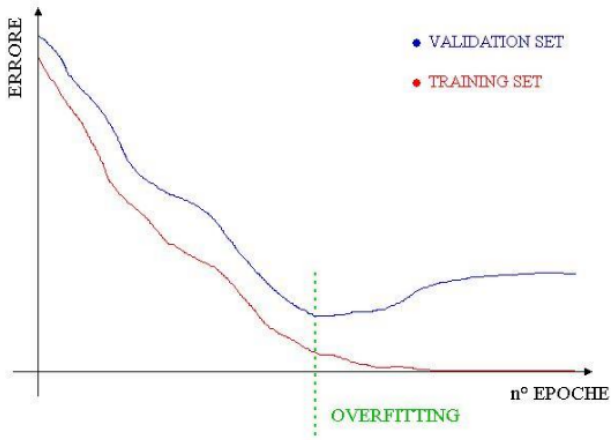


Figure 34: Early stopping.

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.

3 Face Detection

The **face detection** problem is an instance of the general object detection problem, and its goal is to **identify and locate human faces in images or videos**, regardless of their position, scale, pose, orientation illumination etc..

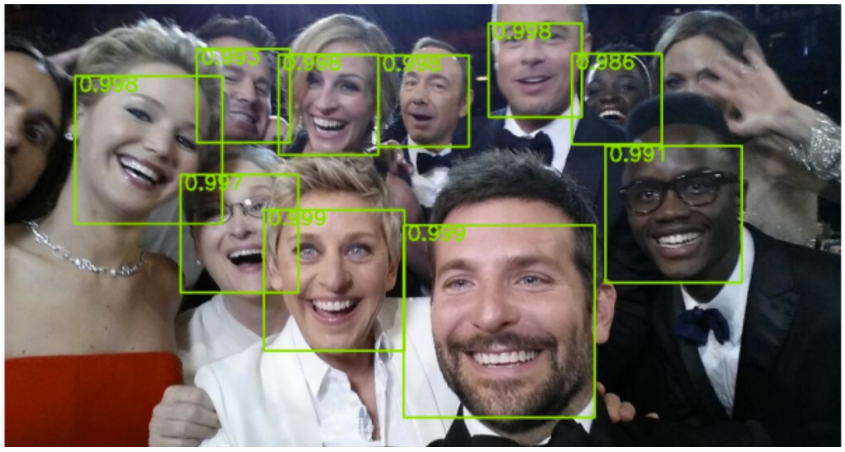


Figure 35: Face detection problem

If we consider a thumbnail 19x19 face pattern, then there are 256^{231} possible combinations of gray values, i.e. 2^{2888}, which is something like 87 times more than the world population, indicating an extremely high dimensional space! In this sense, the main difficulties of this problem rely on:

- the different *scales* of the images;
- the different *poses* of the humans in an image;
- the problem of *occlusion*;
- the different *expressions*;
- *Illumination*.

Moreover, there have been examples of images that fooled a *face detection* algorithm.

Figure 36: Example of image that fools a face detection algorithm

3.1 Related problems

As we introduced before, the *face detection* problem represents an instance of the more general *object detection* problem, and in turn there exist some related problem to the *face detection*'s one:

- *Face localization*, i.e. determine the position of a single face, assuming that the image contains only one face;
- *Facial feature extraction*, i.e. detect the presence and locations of features such eyes, nose etc..;
- *Face recognition*, i.e. compare an input image, called probe, against a database, called gallery, and report a match. Note that this problem is different from face detection, since in the latter we are not asked to recognize the face: in general, *recognition* is concerned with individual identity, while *detection* with the category of an object;
- *Face authentication*, i.e. verify the claim of the identity of an individual in an input image;
- *Face tracking*, i.e. continuously estimate the location and possibly the orientation of a face in an image sequence in real time;
- *Emotion recognition*, i.e. identifying the affective states (happy, sad, disgusted, etc.) of humans;

3.2 Research issues

The main issues of *face detection* problem are:

- **Representation**, i.e. how to describe a typical face;
- **Scale**, i.e. how to deal with faces of different scales;
- **Search strategy**, i.e. how to spot the faces;
- **Speed**, i.e. how to speed up the process of spotting;
- **Precision**, i.e. how to locate the faces precisely;
- **Post-processing**, i.e. how to combine detection results.

3.3 Methods

There exist various methods that are used to address the *face detection* problem, and the most important ones are described below:

- **Knowledge-based methods**: these methods are based on the idea of encoding the human knowledge of what constitutes a typical face (usually the relationships between facial features), and then build a detection algorithm based on these rules. This approach is not related with ML, so it is basically composed of IF-THEN-ELSE;
- **Feature invariant approaches**: in this case the focus is on finding features of a face that are invariant with respect to the pose, the illumination etc.. Still no ML;

- **Template matching methods**: they store several standard patterns to describe the face as a whole or the facial features separately;
- **Appearance-based methods**: these methods exploit the Neural Networks to learn the models (or templates) which capture the representative variability of facial appearance.

In the following sections we will provide an analysis of all the methods described above.

3.3.1 Knowledge-based methods

As we introduced before, the goal of these methods is to encode the relationships between facial features, i.e. the eyes, the nose etc.., in order to build a detection algorithm based on these rules. In this sense, we can recognize a **top-down** approach, since the detection is based on a set of human-coded rules. Some examples of these rules are:

- the center part of face has uniform intensity values;
- the difference between the average intensity values of the center part and the upper part is significant;
- a face often appears with two eyes that are symmetric to each other, a nose and a mouth.

We analyze two different *knowledge-based methods*: the first one was proposed by [Yang and Huang(1994)], while the second one by [Kotropoulos et al.(1994)Kotropoulos, Magnisalis, Pitas, and Strintzis].
Yang and Huang, 1994: this algorithm was based on a multi-resolution focus-of-attention approach. In particular,

- the **Level 1**, i.e. the slowest resolution, applied the first rule described above in order to search for candidates;
- the **Level 2** produced a local histogram equalization, in order to correct the image in case of light issues etc.., followed by edge detection;
- finally, the **Level 3** was responsible for searching the eyes and the mouth features for validation.

Kotropoloulos and Pitas, 1994: the idea of this algorithm was to perform both horizontal and vertical projections of the input image in order to search for candidates. The projections were computed as:

$$HI(x) = \sum_{y} I(x, y)$$

$$VI(y) = \sum_{x} I(x, y)$$

Then, the intersection of these two projections allowed to detect the face. However, it was difficult to detect multiple objects or people in complex background.

In general, the main **advantage** of this method is that it is quite easy to come up with simple rules to describe the features of a face and their relationships, so for this reason this kind of algorithms are simple to be implemented. Moreover, they obtain really good results for face localization in uncluttered background. On the other hand, the main **disadvantage** of this approach is that both detailed and general rules to detect faces may find many false positives, and in general it is difficult to extend this approach to detect faces in different poses.

3.3.2 Feature invariant approaches

While *knowledge-based methods* follow a top-down approach for detecting faces on an image by following some human-coded rules, the *feature invariant methods* follow a **bottom-up** approach. Their goal is indeed to detect invariant facial features (for example the eyes, the nose etc..), then to group them into candidates and, finally, to verify them. Since the features are not random, they can be represented as a graph, where the edges represent the relationships between them: in this sense, the *face detection* problem turns into a *graph matching* problem!
An example of algorithm following this approach was implemented by
[Leung et al.(1995)Leung, Burl, and Perona].
Leung et al, 1995: in his solution, he formulated the *face detection* problem as a problem of finding the correct geometric arrangement of facial features, where the facial features were defined by the average responses of multi-scale filters. Finally, the result was given by a graph matching among the candidates to locate faces.
In general, the main **advantage** of this approach is that, in contrast with the *knowledge-based methods*, in this case the features are invariant to the pose and the orientation changes. On the other hand, two **disadvantages** of this method are that it is difficult to locate facial features due to several corruption (for example, illumination, noise etc..), and to detect features in complex backgrounds.

3.3.3 Template matching models

This approach is similar to the previous one, since its goal is to store one or more hand-coded templates and to use the correlation between the templates and the image in order to locate faces. The templates can be either predefined, i.e. they are based on edges or regions, or deformable, i.e. they are based on facial contours, so they adapt to different faces. Pictures 3.3.3 and 3.3.3 represents, respectively, an example of predefined and deformable templates: as we can see, in both cases the goal of the algorithm is to find some points in the image that match with the template.

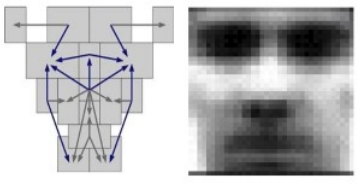


Figure 37: Example of predefined template

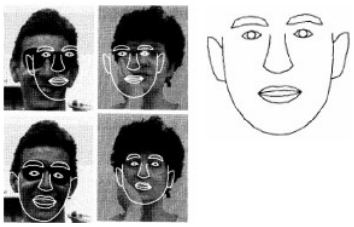


Figure 38: Example of deformable template

In general, these algorithms are simple, but on the other hand they need the templates to be initialized near the face images, and it is difficult to enumerate all the possible templates for different poses (similarly to knowledge-based methods)

3.3.4 Appearance-based methods

The last family of methods we analyze are the *appearance-based methods*, which are the only ones characterized by a ML (or data driven) approach. The general idea of these algorithms is to:

1. Collect a large set of (resized) face and non-face images and to train a binary classifier to discriminate them;
2. Given a test image, they detect faces by applying the trained classifier at each position and scale of the image

Note that the second phase ensures that the algorithms are invariant to different positions and scales. We will analyze three different algorithms: the first one was introduced by [Sung and Poggio(1998)], the second one by
[Rowley et al.(1998a)Rowley, Baluja, and Kanade], while the third one by [Viola and Jones(2001)].
Sung and Poggio, 1994: the main idea of this algorithm is to exploit the sliding window approach. In particular, the sliding window moves across the image running a classifier in order to detect if the portion contains an image or not. Note that this technique is repeated for different sizes of the image, in order to make it invariant to scale. Moreover, as mentioned before, the *appearance-based methods* exploit the functioning of the Neural Networks, and in this case the *back-propagation* technique was exploited. Overall, the algorithm was composed of several phases, which are described below, while Picture 3.3.4 represents an overview of the algorithm.

1. **Pre-processing phase**: in this phase the input image is "cleaned" in the following steps:
    - **Resizing**, i.e. all the image patterns are resized to 19x19 pixels;
    - **Masking**, which reduces the unwanted background noise in a face pattern. This is done by using an oval mask, and not a rectangular one, exploiting the fact that the face in the image does not fill the corner of a rectangular mask;
    - **Illumination gradient correction**, which finds the best fit brightness plane and then subtract from it to reduce heavy shadows caused by extreme lightning angles. This step is done for solving face detection problem for images with shadows;

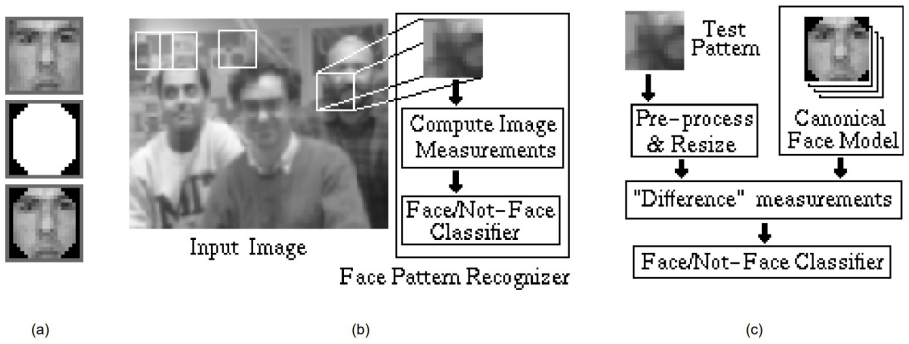


Figure 39: Overview of the algorithm

- **Histogram equalization**, which compensates the imaging effects due to changes in illumination and different camera input gains.

2. **Modeling the distribution of "face" and "non-face" patterns**: now the goal is to classify each portion of the image, and this is done by building a training set composed by both positive examples and negative examples (always 19x19 images). Then the idea was to consider these examples as points, and to cluster these points into **6 different clusters** by using $KMeans$ algorithms, in order to group faces with similar expression (they estimated in 6 the number of different expression of a human face). Each cluster is modeled by a multi-dimensional Gaussian with a centroid and a covariance matrix, and each Gaussian covariance is approximated with a subspace, i.e. by using the largest eigenvectors. A visual representation of this phase is provided in Picture 2.

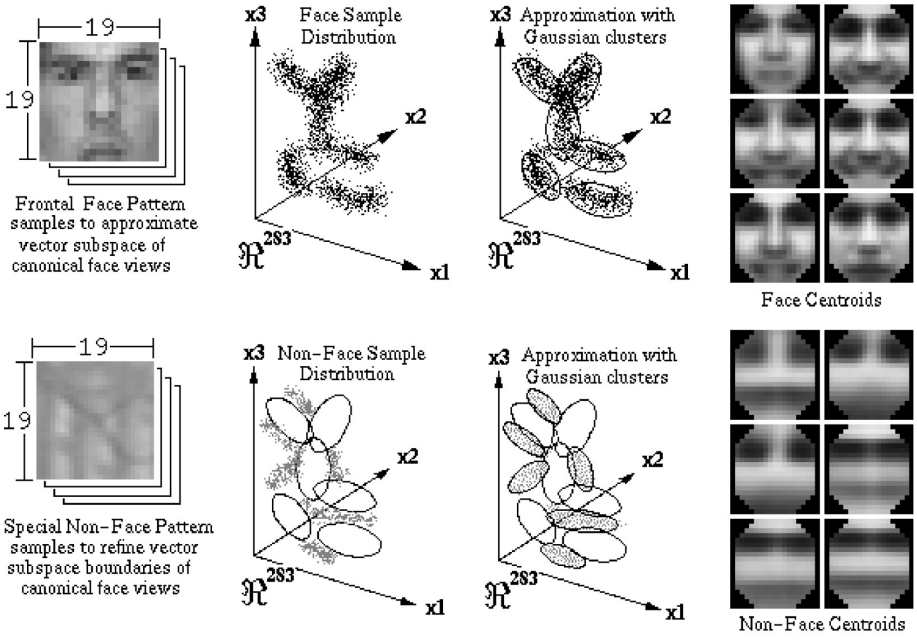


Figure 40: Second phase of the algorithm

3. **Matching patterns with the model**: to detect faces in an input image, the system matches window patterns at different image locations and scales against the

distribution-based face model. Before each match, the system first applies the preprocessing operations to the current window pattern. Each match returns a **set of "difference" measurements** which is fed to a trained classifier that determines whether or not the current window pattern is a frontal face view. More specifically, each set of measurements is a **feature vector of 12 distances** between the normalized input window pattern and the model's 12 cluster centroids in our multidimensional image vector space, as represented in Picture 3.

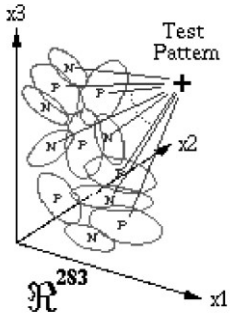


Figure 41: Set of "difference" measurements

Moreover, each of the 12 distances is composed of two parts:

- the *within subspace distance* D_1, which is represented by the Mahalanobis distance between the projected sample and the cluster centroid;
- the *distance to the subspace* D_2, which is represented by the distance between the sample and the subspace.

A visual representation of D_1 and D_2 is provided in Picture 3.

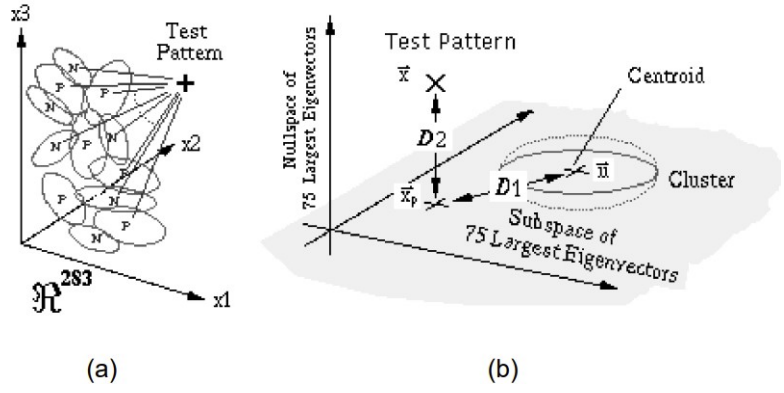


Figure 42: Visual representation of D_1 and D_2

4. **The classifier**: the final phase of the algorithms is the classification phase. A **Multi-layer Neural Network** is used to identify "face" windows patterns from "non face" patterns by using the feature vector of 12 distances derived from the previous phase. The network has:

- 12 pairs of input units, each of which representing a pair (D_1, D_2) of distances;
- 24 hidden units (it was proved that this number does not significantly affect the network's performance);
- 1 output unit ("face" or "non face").

During classification, the net is given a vector of the current test pattern's distance measurements to the 12 cluster centroids: the output unit returns a 1 if the input distance vector arises from a "face" pattern, and a 0 otherwise. The net was trained on feature distance vectors from a database of 47,316 window patterns containing 4,150 positive examples of "face" patterns with a standard **back-propagation learning algorithm** until the output error stabilizes at a very small value.

A crucial aspect of this algorithm is that the feature vector was not chosen from the image, but it was built by using a set of distances from the test patterns: in this sense, the Neural Network worked as a Nearest Neighbor classifier. Another important issue of this method was the **generation of the training set**. As we described above, the training set was composed by both positive and negative examples: for "face" patterns the selection was quite simple, because the simply collected all the possible views of human faces. However, since the collection of "face" patterns was not so large, their idea was to increase it by creating some **virtual examples**, each of which was derived by randomly mirror, rotate, translate and scale the original "face" examples. This technique is called nowadays **data augmentation**. Picture 3.3.4 shows some virtual examples.

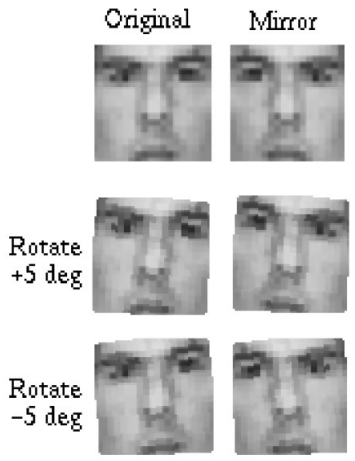


Figure 43: Virtual examples

For "non face" patterns, the task is more tricky. In essence, every square non-face window pattern of any size in any image is a valid training example. Clearly, our set of "non face" patterns can grow intractably large if we are to include all possible "non face" patterns in the training set. To constrain the number of "non face" examples in the training set, they used the following **bootstrap** strategy that incrementally selects only those "non face" patterns with high utility value with the following steps:

1. Start with a random small set of "non face" examples in the training set;

2. Train the Neural Network classifier with the current training set;
3. Run the learned face detector on a sequence of random images;
4. Collect all the "non face" patterns that the current detector wrongly classified as faces, i.e. the false positives;
5. Add these "non faces" patterns to the training set;
6. Repeat from 2 until satisfied.

Note that this technique leads to two important consequences: the first one is that it provides a simple method to build "non faces" patterns, and the second one is that the training set is enlarged with examples that steer the classifier away from its current mistakes.
Finally, one last feature of this algorithm is that in order to be invariant to different scales, the input images were downsampled by a certain scaling factor. Picture 3.3.4 shows a result of the algorithm.

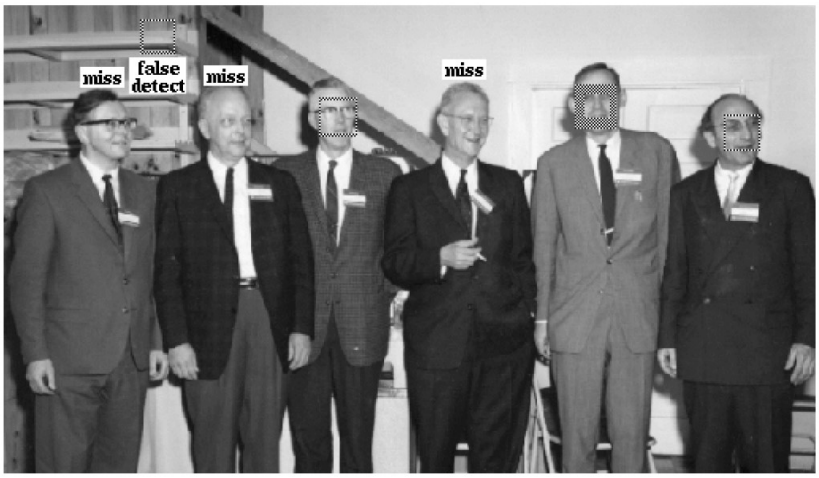


Figure 44: Results of the algorithm

Rowley-Baluja-Kanade (1996): this algorithm is much similar to what we do nowadays, and the main difference with the previous one relies on the way in which the Neural Network is used, while the pre-processing and the other phases are quite similar. Picture 3.3.4 represents an overview of the algorithm.
As we can see, the pre-processing phase is very similar to the previous algorithm, with the only difference with the size of the patterns, which is now 20x20 (the previous one was 19x19). As we said before, the main difference relies in the **Neural Network**: in this case the features were extracted directly from the image, and they exploited the concept of **receptive field** of a neuron. While in a standard Neural Network, each neuron receives in input all the outputs of the neurons in the previous layer, in a Neural Network with receptive fields, each hidden unit receives in input the outputs of a subset of neurons of the previous layer, as showed in Picture 3.3.4.
In this sense, in this algorithm the input patch was splitted in several sub-regions, each of which was "controlled" by different neurons. The size of the receptive fields of the hidden unit were different, in order to detect local features that could be important for

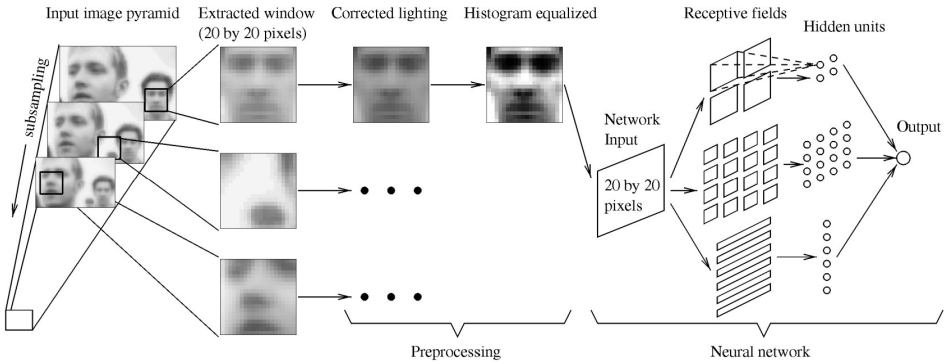


Figure 45: Overview of the algorithm

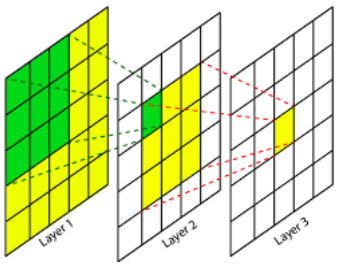


Figure 46: Representation of the functioning of the receptive field of a neuron

the problem of face detection. In particular, the hidden units with horizontal receptive fields were able to detect such features as mouths or pairs of eyes, while the hidden units with square receptive fields were able to detect features such as individual eyes, the nose, or corners of the mouth.
The overall performances of this algorithm were better than Sung and Poggio, and in general their results were astonishing at that time. In Picture 3.3.4 we provide an example of result of this algorithm.

Figure 47: Results of the algorithm

However, in 1998 the same authors provided a new version of the algorithm ([Rowley et al.(1998b)Rowley, Baluja, and Kanade]) which was able to detect **rotated**

faces. The idea for solving this new problem was the following one: before feeding the face detector, they built a Neural Network, called *Router Network* which was trained to estimate the rotation angle of the input window. If the window contained a face, the the *Router* returned the angle of the face and the face was rotated back to upright frontal position, otherwise it returned a meaningless angle. Then, the de-rotated window was applied to the face detector, which was previously trained for upright frontal faces. Picture 3.3.4 represents an overview of this second version of the algorithm.

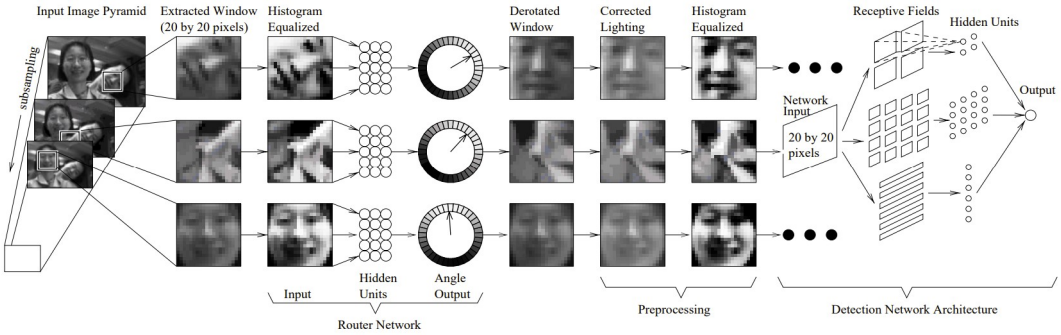


Figure 48: Overview of the new version

In order to implement these operations, the training set for the *Router Network* was built by exploiting the so-called **self learning** technique, which avoids to manually label the whole training set. The value of the rotation angle for each image was not manually added, but it was retrieved by rotating the original upright image by a certain value, which therefore was added in the training set.
Despite being much more accurate that Sung and Poggio, this algorithm had still some problems with the accuracy, and, especially, it had unacceptable CPU times.
Viola and Jones, 2001: this algorithm represented the first real-time face detector, and for this reason it was implemented in real systems for a long time. It was characterized by a slow training phase, while the detection phase was very fast; in general, the three main contributions of this work were:

1. A new image representation called **integral image**, which allows the features used by the detector to be computed in a very fast way;
2. A learning algorithm, based on *AdaBoost*, for feature selection: it selects a small number of critical visual features from a larger set and yields extremely efficient classifiers;
3. A method, called **attentional cascade**, for fast rejection of non face windows. The idea was to use sliding windows, but without wasting any time in a non face portion of the image.

Now we focus on each of the three contributions. In general, the object detector was based on very **simple features**: they used *two-rectangle features*, *three-rectangle features* and *four-rectangle features*, as represented in Picture 3.3.4.
Each of the features computed the sum of the values in the white area subtracted by the sum of the values in the black area. In this way, the value of the feature was maximum

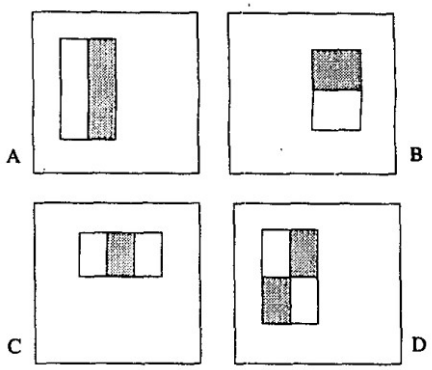


Figure 49: Rectangular features

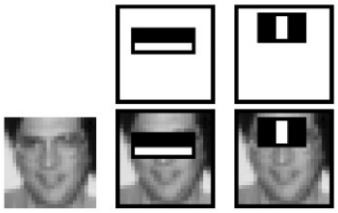


Figure 50: Features and an image

if all the relevant pixels of the underlying portion of the image were positioned under the white area of the rectangle. An example is represented in Picture 3.3.4.
Moreover, the rectangle features can be computed very quickly using an intermediate representation of the image that is called **integral image**. The *integral image* computes a value at each pixel (x, y) as the sum of the pixel values above and to the left of (x, y) inclusive, i.e.

$$ii(x, y) = \sum_{x' \leq x, y' \leq y} i(x', y')$$

, where $i(x', y')$ corresponds to the original image. An important property of the *integral image* is that it can be computed in a single pass over the image, i.e. in linear time, by considering the following pair of recurrences:

$$s(x, y) = s(x - 1, y) + i(x, y)$$

$$ii(x, y) = ii(x, y - 1) + s(x, y)$$

, where $s(x, y)$ represents the cumulative sum of the row values. In this sense, we can exploit this important property in order to compute in linear time the sum of the values of the rectangular features. For example, if A,B,C and D are the values of the *integral image* at the corners of a rectangle, then the sum of the original image values in the rectangle is given by A-B-C+D. As we can see, only three additions are required for any size of the rectangle, which make the feature extraction a very fast operation. Notice that for a 24x24 detection region, which was the region used in the model, the number of possible rectangle features is around 180,000.

Then, the second important innovation of the algorithm was about the **learning algorithm** adopted both for the feature selection phase and to train the classifier, which was based on the **ensemble approach**. While in a standard ML pipeline a single complex learning algorithm is trained with the training set and tested with the test set, in the *ensemble approach* a family of weak algorithms is trained, and the prediction is taken from the majority vote or the average of the predictions of all the weak algorithms. Note that a "weak learner" needs only to do better than chance, i.e. better than a model that predicts by tossing a coin. The functioning of this approach is represented in Picture 3.3.4.

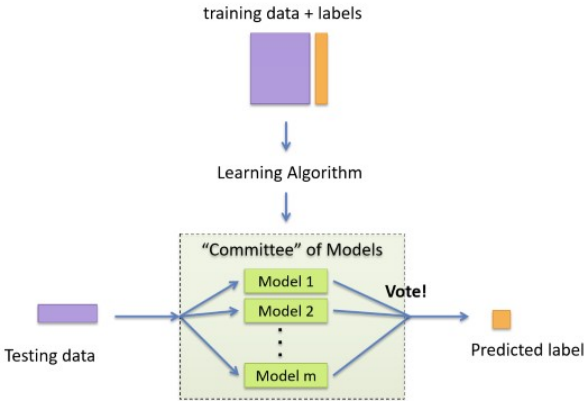


Figure 51: Functioning of the ensemble methods approach

One example of algorithms exploiting this idea of the *ensemble methods* is the **Boosting** algorithm: in this case each weak model is trained sequentially, and the instances that are misclassified by a model are given more weight, i.e. more attention, so that the following model focuses more on classifying correctly those instances.

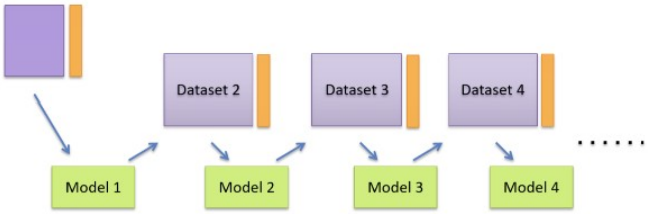


Figure 52: Boosting algorithm

In particular, the *Boosting* algorithm can be summarized as follows:

1. Initially, all the weights of the training set are equal;
2. In each boosting round, the weak learner that achieves the lowest weighted error in chosen, and the weights of the training examples that are misclassified by this learner are risen;
3. The final classifier is computed as a linear combination of all weak learners, where the weight of each learner is directly proportional to its accuracy.

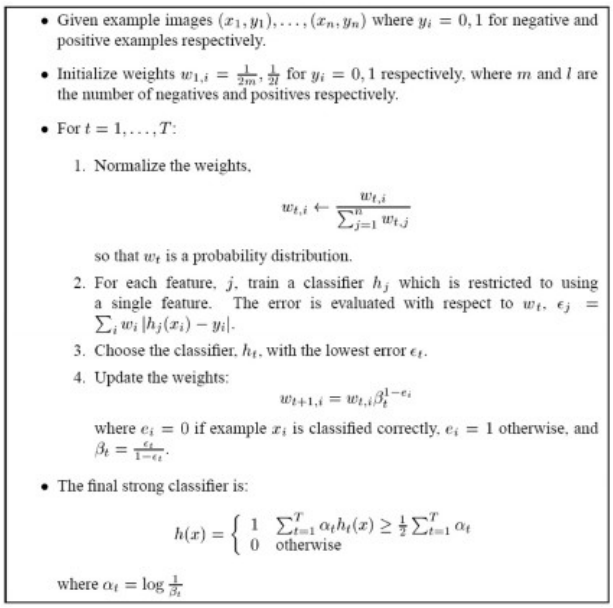
The exact formulas for re-weighting and combine the weak learners depend on the particular boosting scheme (e.g. *AdaBoost*).
As we said before, even though each of the rectangular feature could be computed very efficiently by exploiting the *integral image* representation, computing the complete set of 180,000 features was prohibitively expensive. Their hypothesis was that a very small number of these features could be combined to form an effective classifier, so the *feature extraction* phase was implemented using the *ensemble approach* define above. More specifically, for each feature, the weak learner determines the optimal threshold classification function, such that the minimum number of examples are misclassified. From a mathematical point of view, the learner is defined as:

$$h_j(x) = \begin{cases} 1 & \text{if } p_j f_j(x) < p_j \theta_j \\ 0 & \text{otherwise} \end{cases}$$

, where:

- x is a 24x24 window of the image;
- p_j is a parity bit, i.e. it is equal to 1 if we impose the feature to be greater than the threshold, -1 otherwise;
- $f_j(x)$ is the response of the rectangle feature, computed using the *integral image*;
- θ_j is the thershold.

Finally, Picture 3.3.4 represents the *AdaBoost* algorithm which was implemented for both feature selection and training the classifier.



- Given example images $(x_1, y_1), \ldots, (x_n, y_n)$ where $y_i = 0, 1$ for negative and positive examples respectively.
- Initialize weights $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ for $y_i = 0, 1$ respectively, where m and l are the number of negatives and positives respectively.
- For $t = 1, \ldots, T$:
  1. Normalize the weights,
     $$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$
     so that w_t is a probability distribution.
  2. For each feature, j, train a classifier h_j which is restricted to using a single feature. The error is evaluated with respect to w_t, $\epsilon_j = \sum_i w_i |h_j(x_i) - y_i|$.
  3. Choose the classifier, h_t, with the lowest error ϵ_t.
  4. Update the weights:
     $$w_{t+1,i} = w_{t,i} \beta_t^{1-e_i}$$
     where $e_i = 0$ if example x_i is classified correctly, $e_i = 1$ otherwise, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$.
- The final strong classifier is:
  $$h(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases}$$
  where $\alpha_t = \log \frac{1}{\beta_t}$

Figure 53: AdaBoost algorithm

Some notes:

- The training set is composed of both positive and negative examples;

- The weights are initialized uniformly;
- Point 2. computes the error of the learner w.r.t. the feature j: notice that $|h_j(x_i) - y_i|$ represents the actual error of the learner, while w_i represents the weight of the example;
- Point 4. updates the weights of the examples: notice that $\beta_t < 1$, so if the example is correctly classified, then its weight is reduced, otherwise it remains the same. However, since the weights are normalized, this reduction will lead to an increase of the weights of the misclassified examples.

The first feature that was selected seemed to focus on the property that the region of the eyes is often darker than the region of the nose and cheeks , while the second selected feature relied on the property that the eyes are darker than the bridge of the nose. This feature combination could yield almost 100% detection rate and 50% false positive rate. Moreover, it was proved that a 200-features classifier could yield to a 95% detection rate with a false positive rate of 1 in 14084; however, adding features to the classifier directly increases the computation time.
The last crucial idea developed in this model was the method of **attentional cascade**, whose goal was to construct a cascade of classifiers which achieves increased detection performance while radically reducing computation time. The key idea is that smaller, and therefore more efficient, boosted classifiers can be constructed in order to reject many of the negative sub-windows while detecting almost all positive instances. Simpler classifiers are used to reject the majority of sub-windows before more complex classifiers are called upon to achieve low false positive rates. A visual representation of this method is provided in Picture 3.3.4.

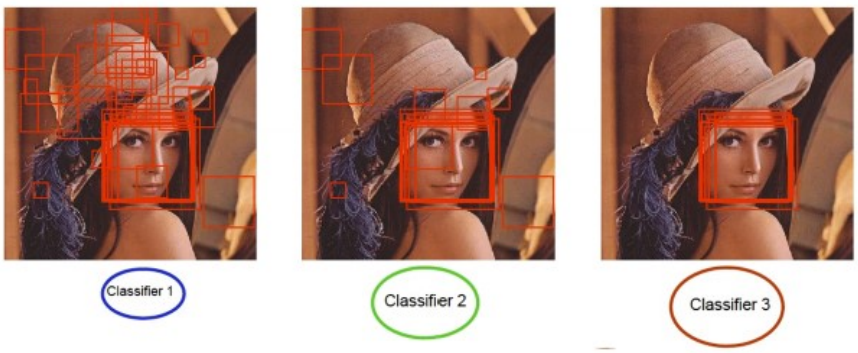


Figure 54: Attentional cascade

As we can see, the first classifier sees all the possible sub-windows, but a negative outcome at any point leads to the immediate rejection of the sub-window, which is therefore not considered by the following, a more complex classifier. In the case of the *attentional cascade* method, both the detection rate and the false positive rate are computed by multiplying the respective rates of the individual stages: in general, it was proved that a detection rate of 0.9 and a false positive rate of 10^{-6} can be achieved by a 10-stage cascade if each stage has a detection rate of 0.99 and a false positive rate of 0.30. Two possible problems about the *attentional cascade* approach can be either that a classifier rejects a sub-window in which there's a face or that several detections are found for a single

face. In particular, the second issue is shared among all the appearance-based methods we discussed, and it can be solved by applying the *non-maximum suppression* method, which is implemented as follows:

- The set of detections are first partitioned into disjoint subsets: two detections are in the same subset if their regions overlap;
- Each partition yields a single final detection, whose corners are the average of the corners of all detections in the subset.

A visual result of the application of the *non-maximum suppression* method is provided in Picture 3.3.4.

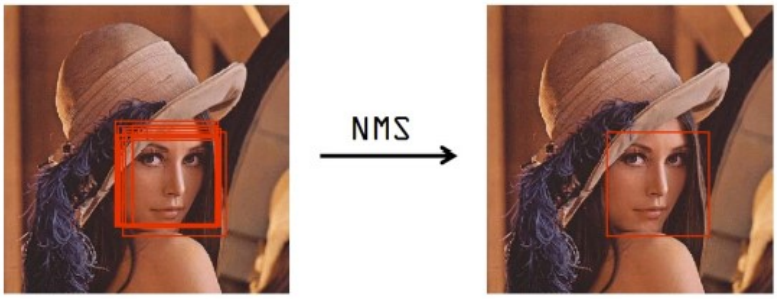


Figure 55: Result of non-maximum suppression process

The *Viola and Jones* algorithm was trained with 4,916 hand labeled faces and 10,000 non faces, containing many variations (illumination, pose etc..). The training phase took several weeks, and the final detector contained 38 layers for a total of 6,061 features. The test time was 15 times faster that *Rowley-Baluja-Kanade*.

4 Human Detection

The goal of the **human detection** problem is to detect and localize persons in an image regardless of their position, scale, pose, orientation and illumination.

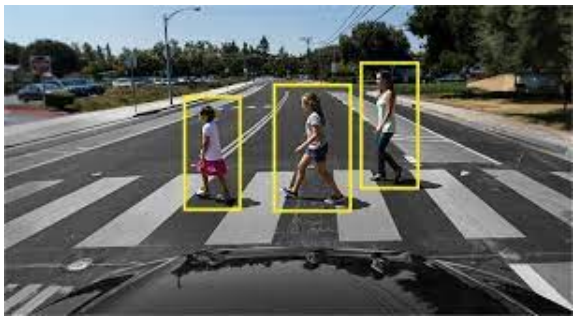


Figure 56: Pedestrian recognition problem

The main difficulties of this problem are:

- The wide variety of articulated *poses*;
- Variable *appearance and poses*;
- Complex *background*;
- *Occlusion*;

4.1 Research issues

The main issues of *pedestrian recognition* problem are very similar to the *face detection*'s ones:

- **Representation**, i.e. how to describe a typical person;
- **Scale**, i.e. how to deal with persons with different sizes;
- **Search strategy**, i.e. how to spot the persons;
- **Post-processing**, i.e. how to combine detection results.

4.2 Method

The usual method for solving the problem of *pedestrian recognition* is given by the *sliding window* approach, in particular by following these steps:

1. **Inspect** every window of the image at all scales and locations;
2. Given a window, extract a **feature vector**, i.e. a vector of numbers that describes the window's contents;
3. **Classify** each feature vector and accept a window if the score is above a certain threshold;
4. **Post-processing** phase in which the mess is cleaned-up.

In the following sections we describe in detail each of the phase.

4.2.1 Inspection phase

In this phase, the **sliding window** scans the whole image at all scales and locations: usually, the window is 128 pixels tall and 64 pixels wide, as represented in Picture 4.2.1. This 2-to-1 aspect ratio is a rough compromise between the aspect ratio of a person viewed from the front and one viewed from the side with legs fully extended during a step.

Figure 57: Sliding window

Moreover, in order for the *pedestrian detector* to detect persons of very different scales in the same image, the window also scans a down-scaled version of the image, while the size of the window remains the same. This operation is repeated for many down-scaled version of the image, where the scale differences across the scans are small.

4.2.2 Feature vector

The second phase of the *pedestrian recognition* solving method is given by the *feature vector* creation. Since the method which is used for creating each feature vector is based on another family of methods, we first introduce the latter methods in order to better understand the former one.

Support Vector Machines (SVM)

One of the most important family of ML algorithm is represented by the **Support Vector Machines (SVM)**, which were introduced after the diffusion of the *Backpropagation* algorithm, and which are very connected to *Statistical Learning Theory*. *SVM* deals with binary classification problems (with the assumption that the problem is linearly separable), and it 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. More specifically, the goal of *SVM* is to find, among all the possible class boundaries between the two classes, the one that is furthest from the two classes. If the distance between the closest points of two different classes is called **margin**, then *SVM*'s goal is to separate the instances of the two classes with a boundary that **maximizes the margin**: intuitively, the farthest the boundary is from the points, the greater the confidence in the prediction is. Picture 4.2.2 provides a visual representation of the margin: the instances of the two classes that determine the margin, i.e. the red circled points, are called **support vectors**. In this sense, we can easily notice that decision boundary only depends on the support vectors, since they're the only points that determine the width of the margin.

If we denote as $w^t x + b = 0$ (or, equivalently, as $c(w^t x + b) = 0$) the plane which separates the instances of the two classes, i.e. the decision boundary, then we can normalize the

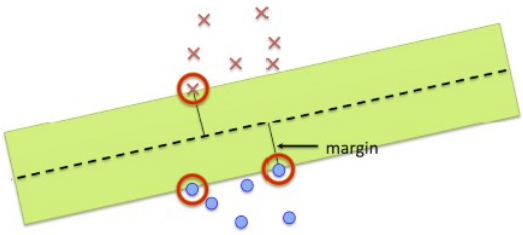


Figure 58: Margin and support vectors

points of the dataset so that the *positive* support vectors, i.e. the support vectors *above* the decision boundary, lie on this plane:

$$w^t x + b = +1$$

, while the *negative* support vectors on this one:

$$w^t x + b = -1$$

In this way, the distance between each support vector and the decision boundary is given by $\frac{1}{||w||}$, so the **equation of the margin** is $\frac{2}{||w||}$. A visual representation of these relationships is provided in Picture 4.2.2.

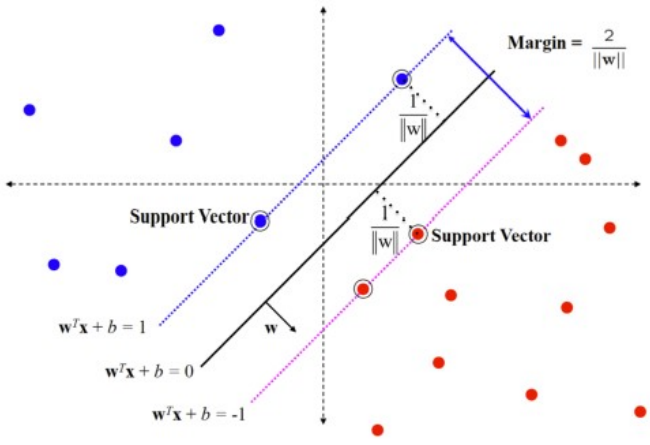


Figure 59: Margin and support vectors

As we said before, the goal of SVM is to maximize the margin, so learning the SVM can be formulated as the following optimization problem:

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

The formulation of 1 represents an optimization problem on a convex function and with only linear constraints. Due to the nature of the problem we have a **unique minimum** solution which corresponds to the **optimal margin classifier**.
One of the issues of SVM is to make decisions in presence of outliers. The previous image

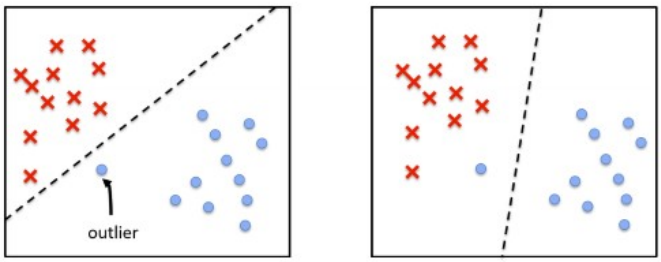


Figure 60: SVM in presence of outliers

shows the effect that a single outlier can have. We can make only one of the following choices: correctly classify the outlier, thus reducing the robustness of the classifier, or leave the outlier as misclassified. The strategy which is commonly adopted is the second one, as it provides more generalization to the classifier. In order to reduce the effects of mis-classification we introduce some **slack variables**, useful for allowing some errors in the boundary. More specifically, a slack variable is created for each example of the training set, and each slack variable indicates how much the constraints are violated. For example, Picture 4.2.2 shows an example of examples of the training set and the respective slack variables: in one case the value of the slack variable is 0.6, indicating that the training instance is violating the constraints of the boundary but it still classified correctly since the value is smaller than 1. In the second case, the value is 1.3, which means the instance is wrongly classified.

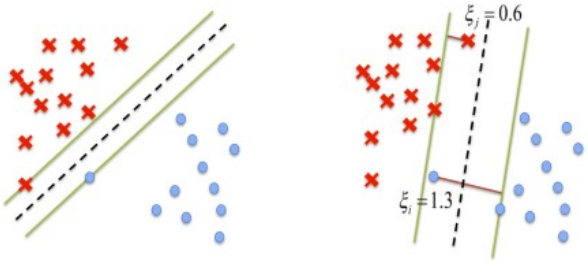


Figure 61: Slack variable example

Now the optimization problem can be re-formulated as:

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

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)*, allowing more misclassifications.

- large C makes constraints hard to ignore *(narrow margin)*, allowing less misclassifications.
- $C = \infty$ enforces all constraints *(hard margin)*, so in this case no misclassification occurs.

Histograms of Oriented Gradients (HOG)

We can now focus on the method which was used to extract the feature vector from each window. The method is called **Histograms of Oriented Gradients (HOG)**, and it was introduced by [Dalal and Triggs(2005)]. As we said before, its goal is to create a robust feature set that allows the human form to be discriminated cleanly, even in cluttered backgrounds under difficult illumination. Picture 4.2.2 provides an overview of the feature extraction chain.

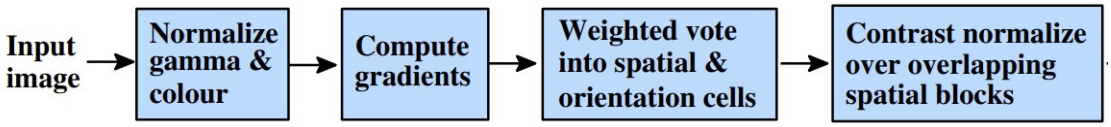


Figure 62: Overview of HOG method

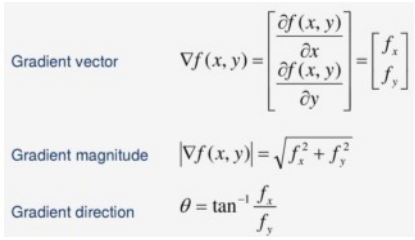
The basic idea is that local object appearance and shape can often be characterized rather well by the **distribution** of local intensity **gradients** or edge directions, even without precise knowledge of the corresponding gradient or edge positions. In this sense, we can notice that the method is based on the concept of gradient, and in particular on the distribution of gradients.
If we have a 1-variable function the gradient is computed as:

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x) - f(x - \Delta x)}{\Delta x} = f'(x)$$

Suppose that the function is discretized, i.e. it is obtained by sampling a continuous function, then the approximated gradient is given by:

$$\frac{df}{dx} = \frac{f(x) - f(x-1)}{1} = f(x) - f(x-1) = f'(x)$$

, i.e. the approximated gradient is computed as the difference of two consecutive discrete points.
If we consider a 2-variables function, then the following relations hold:



, so for computing each component $\frac{\delta f(x,y)}{\delta x}$ and $\frac{\delta f(x,y)}{\delta y}$ an approximation can be used. *HOG* method uses the following computation:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

Therefore, in this case if we have a point centered in $(0, 0)$, then $\frac{\delta f(x,y)}{\delta x}$ = [-1, 0, 1], while $\frac{\delta f(x,y)}{\delta y} = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$

As an example, the gradient of the pixel in Picture 4.2.2 is given by $(94 - 56), (93 - 55)$, i.e. $(38, 38)^T$, while the angle is $arctan(\frac{38}{38}) = 45$ degrees.

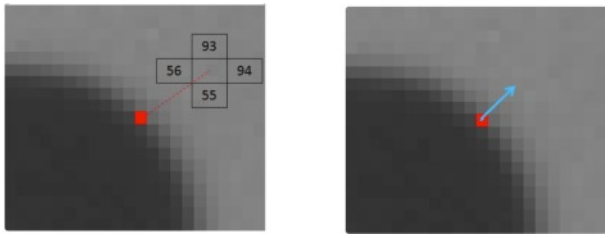


Figure 63: Example of computation of the gradient and its direction

Using the gradient distribution, we can compute the gradient magnitude and direction, i.e. the angle, for each of the pixels in the image, as shown in Picture 4.2.2.

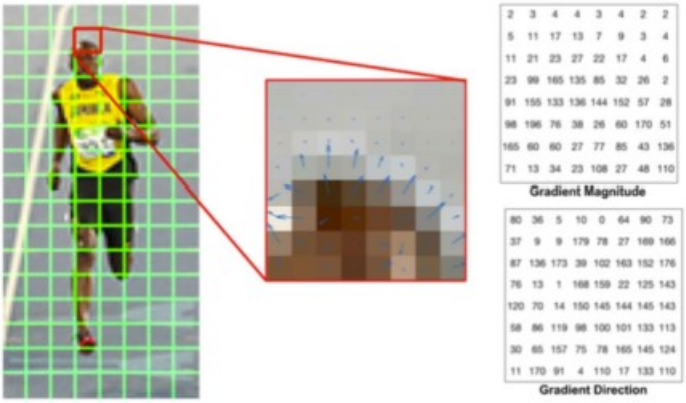


Figure 64: Example of computation of the gradient and its direction for each pixel of the image

Once we introduced this method for gradient computation, we can now analyze how it is used for the task of feature extraction implemented by *HOG*. This method follows these steps:

1. Compute centered horizontal and vertical gradients with no smoothing;
2. Compute gradient orientation and magnitudes (for color images, pick the color channel with the highest gradient magnitude for each pixel);

3. Divide the 64x128 image into 16x16 blocks with 50% of overlap (for a total of 7x15 = 105 blocks). Each block should consist of 4 cells with size 8x8, in which the gradient is computed. A visual representation of the blocks and the cells is provided in Picture 3.

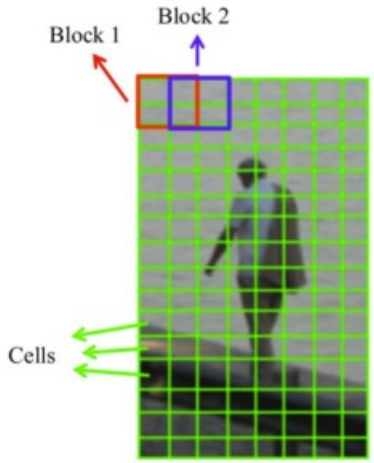
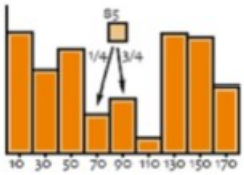


Figure 65: Partition of the image into cells and blocks

4. For each 8x8 cell, compute the histogram of the gradient orientation binned into 9 bins: the vote of the cell is given by the gradient magnitude, and each vote is linearly interpolated between neighboring bin centers. For example, if the orientation is 85 degrees, then the distances to the bin center 70 and 90 are 15 and 5 degrees respectively. Hence, the vote is splitted into $\frac{5}{20} = 0.25$ for Bin 70, and $\frac{15}{20} = 0.75$ for Bin 90.



   Note that the vote could be also weighted with a Gaussian function in order to downweight the pixels near the edges of the block.

5. Concatenate the 4 cell histograms in each block into a single block feature f and normalize f by its Euclidean norm, i.e.:

$$f = \frac{f}{\sqrt{||f||_2^2 + \epsilon^2}}$$

   , where ϵ is used to have better performances.

6. Finally, the final feature vector is composed of 3,780 histograms.

4.2.3 Classification phase

Once the feature extraction phase is done, the classification can be performed. This classification phase was composed by:

- a **learning phase**, in which an SVM classifier was trained with the feature vectors resulted by the HOG method;
- a **test phase**, in which the classifier was asked to predict the presence or the absence of a person in each window in the image.

Since this new approach gave near-perfect separation on the original MIT pedestrian database (507 positive windows only training set, 200 positive windows only test set), the authors introduced a more challenging dataset containing over 1,800 annotated human images with a large range of pose variations and backgrounds. The training dataset was composed of 1,208 positive windows examples and 1,218 negative windows examples, generated using the *Bootstrapping* technique, while the test set by 566 positive examples and 453 negative examples.

4.2.4 Post-processing phase

As we saw for Viola and Jones, the algorithm resulted in multiple responses around positive examples, so the *non-maximum suppression* technique was performed, in order to remove all the boxes that overlapped more that 50% with another box.

5 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 66: 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.

5.1 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.

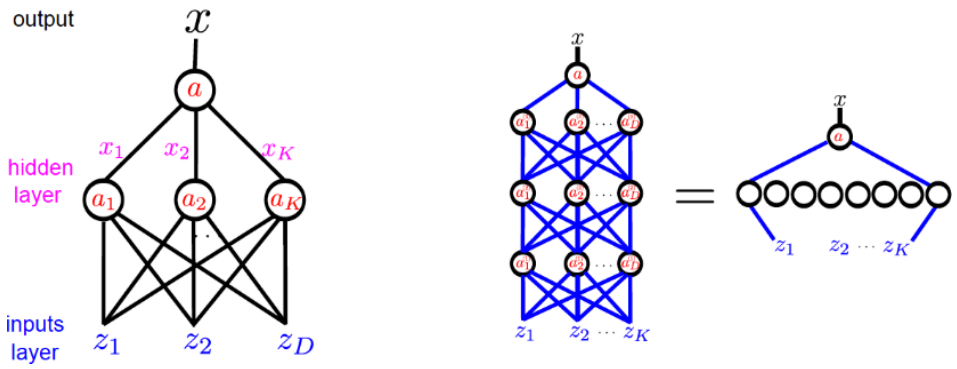


Figure 67: Shallow vs Deep Networks.

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

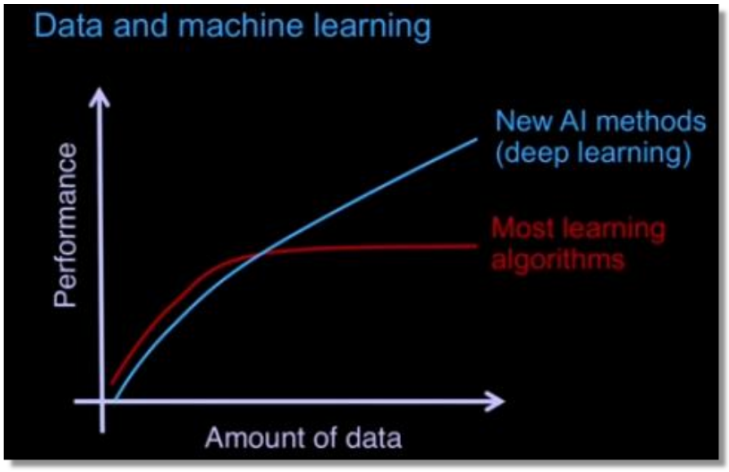


Figure 68: Performances improves with more data.

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.

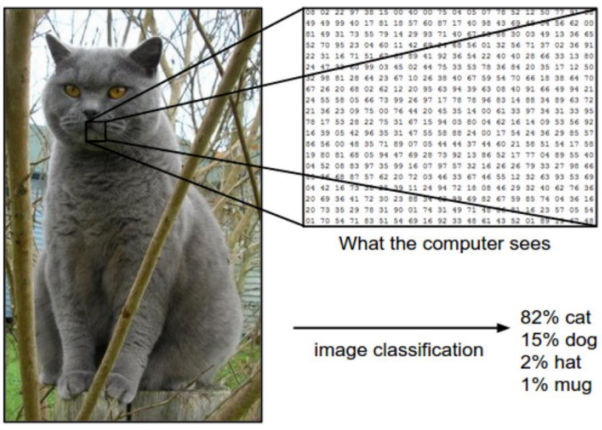


Figure 69: Image classification example.

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

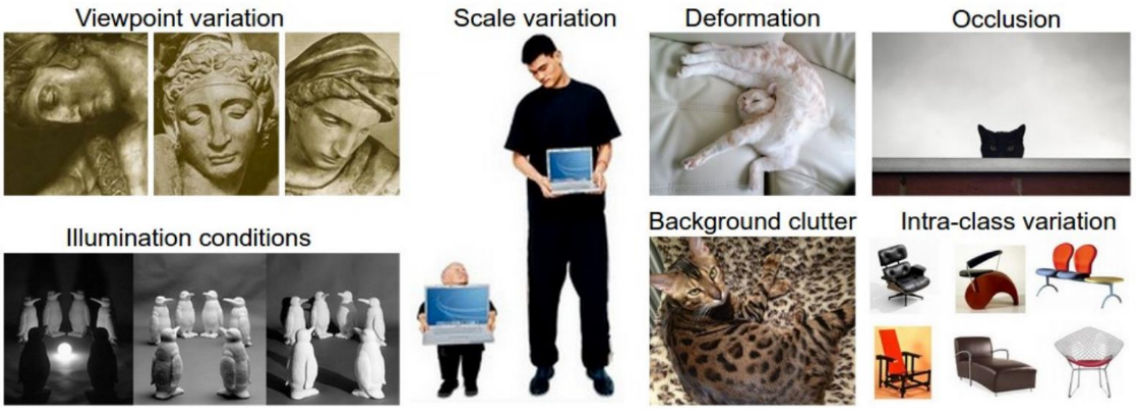


Figure 70: Challenges in image classification.

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.

5.2 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

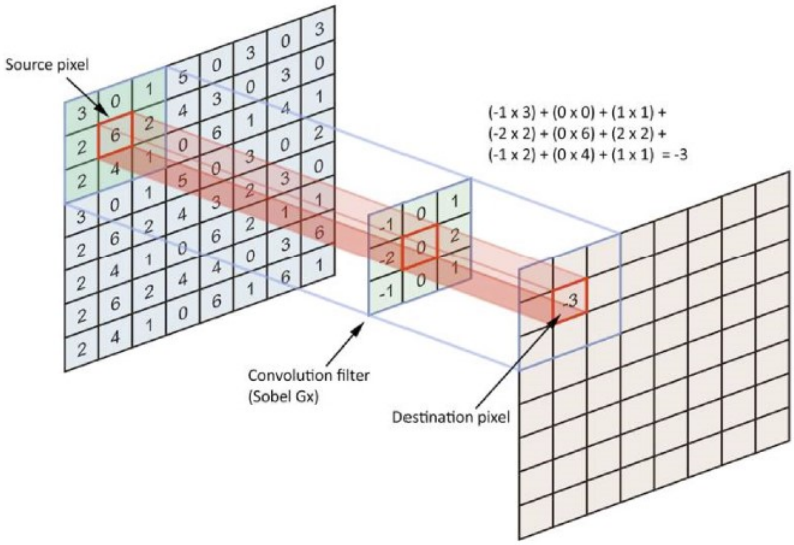


Figure 71: Example of convolution

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.

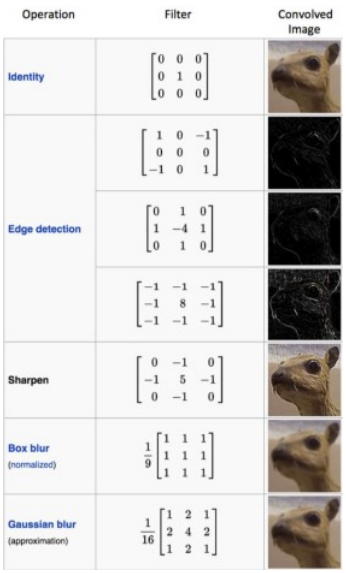


Figure 72: Types of filters

5.2.1 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.

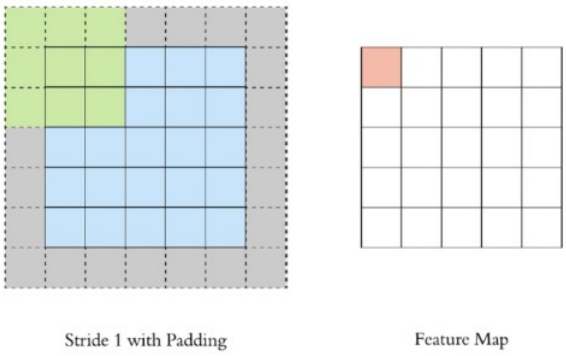


Figure 73: Stride = 1 with padding = 1

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

5.2.2 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.

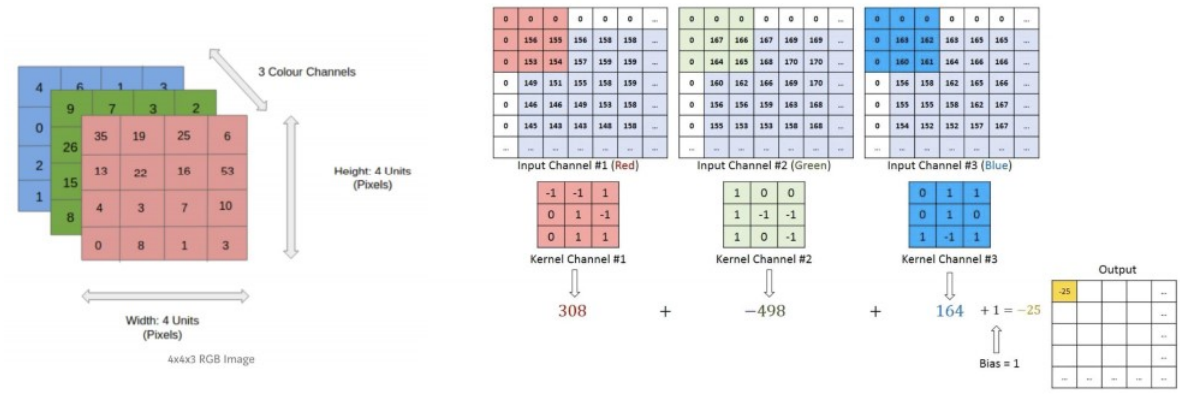


Figure 74: Example of convolution of an image with multiple channels

5.2.3 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 75: Example of 7x7 Gaussian filter

5.2.4 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 76: 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..

5.3 Convolutional Neural Networks (CNNs)

5.3.1 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.

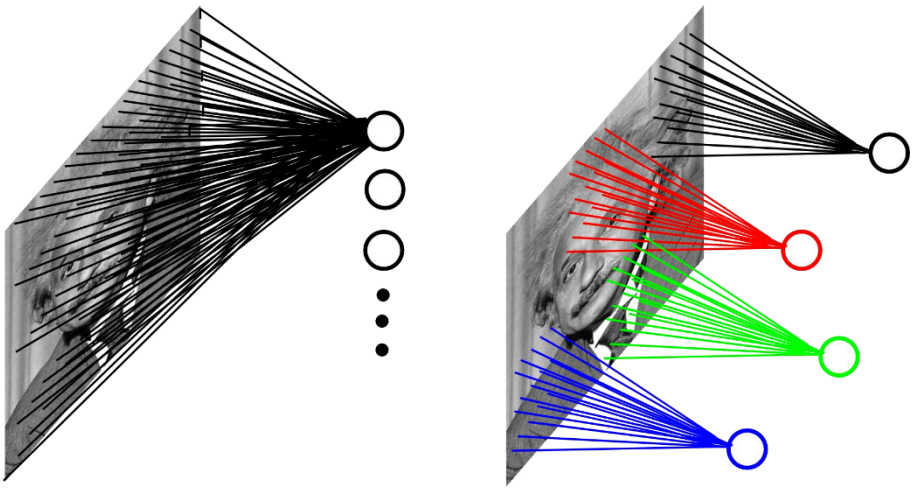


Figure 77: Fully vs local-connected networks.

5.3.2 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.

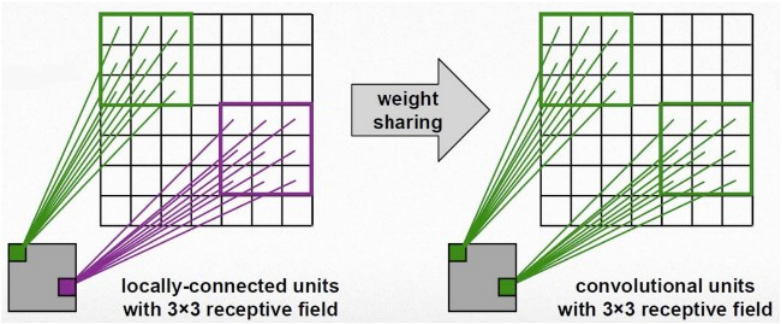


Figure 78: Edge detection problem

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.

5.3.3 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.

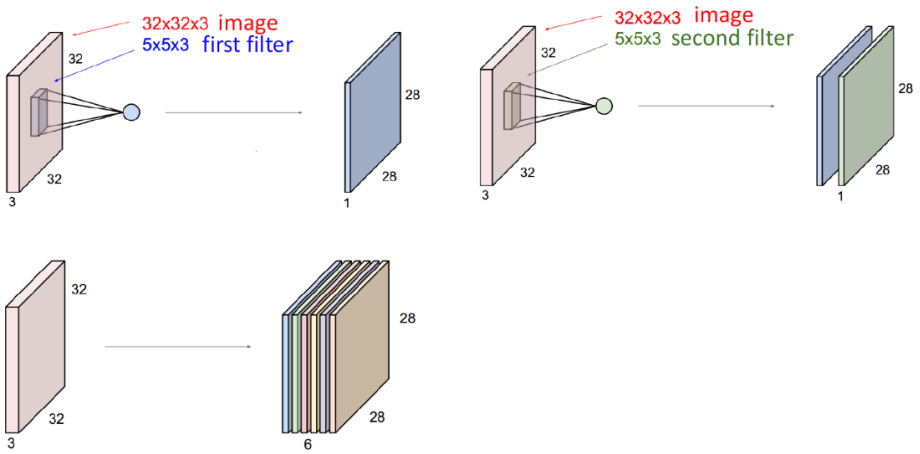


Figure 79: Using Several Trainable Filters.

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.

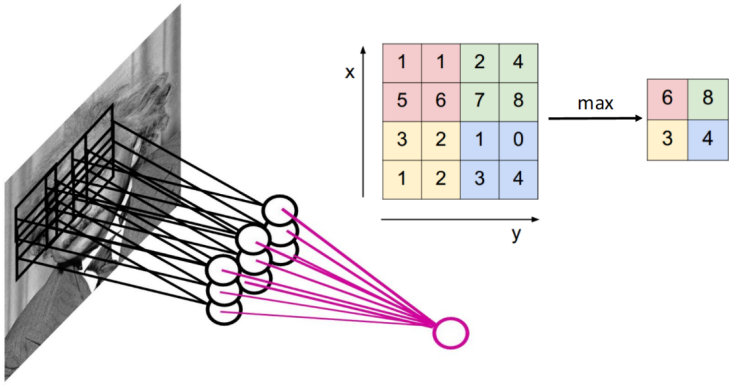


Figure 80: Max Pooling.

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.

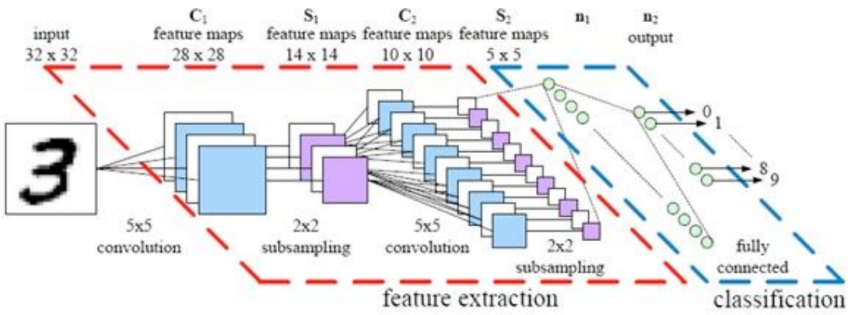
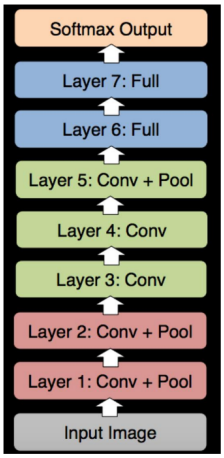


Figure 81: Combining Feature Extraction and Classification.

5.4 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.

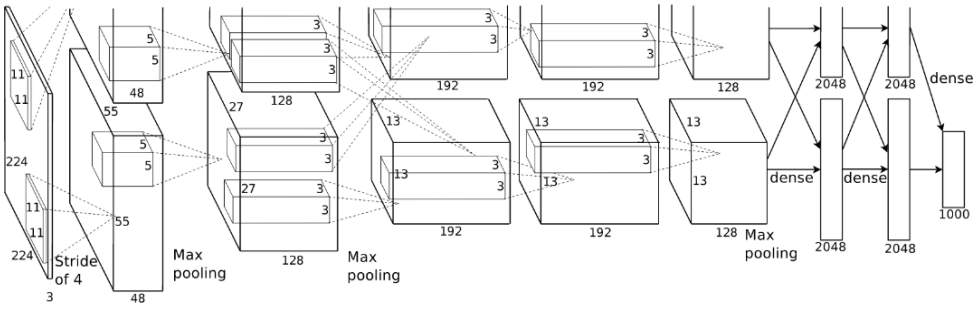


Figure 82: AlexNet architecture.

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.

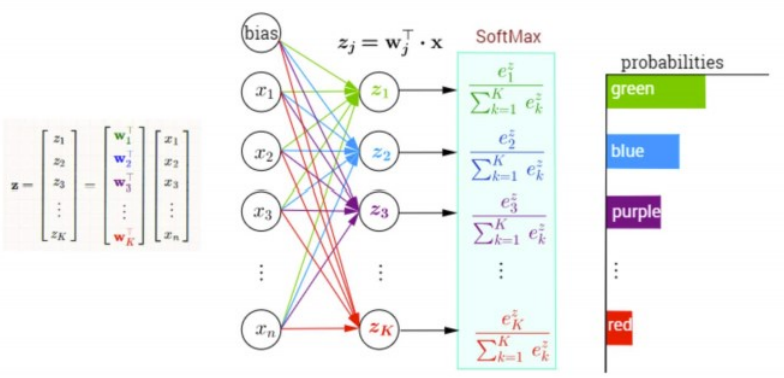


Figure 83: Output of the SoftMax function

5.5 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.

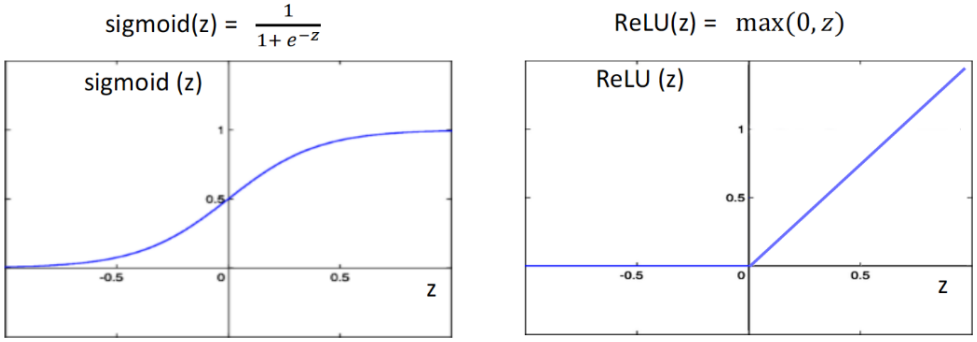


Figure 84: Comparison between sigmoid and ReLU functions.

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**.

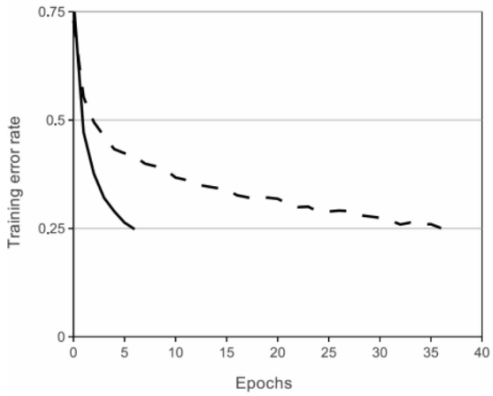


Figure 85: Convergence of ReLU neurons.

5.6 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.

5.7 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 86: Data augmentation: translation, reflection, scaling, rotation, RGB noise.

5.8 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.

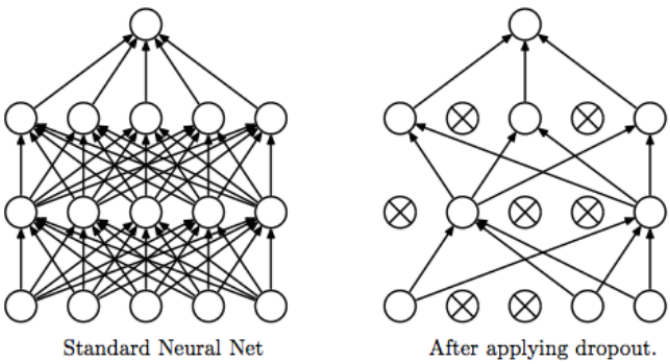


Figure 87: Dropout results

5.9 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 88: 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.

5.10 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. An application of this kind of problem can be found in the autonomous driving systems and in medical image diagnostics;
- **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.

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

5.10.1 Semantic segmentation

As we introduced before, the goal of semantic segmentation is to assign a semantic label to each and every pixel of the image, and we can take into consideration two important features:

- The contextual information is quite useful in this problem;
- We cannot exploit the architecture of AlexNet for solving this problem, since AlexNet provides a probability distribution. However, we can exploit the part of the network responsible for the feature extraction phase.

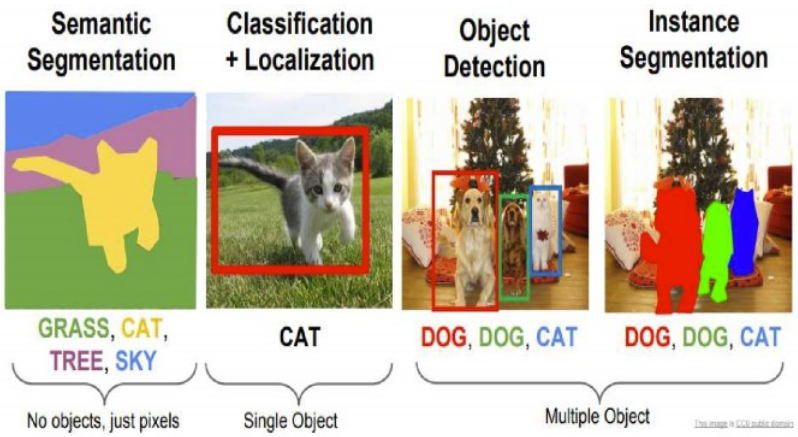


Figure 89: Computer vision problems

First idea: sliding window The first method for solving this problem is represented by the **sliding window** approach: in this case a small **patch** is moved along the image, and each **center pixel** of the patch is **classified** using a **CNN**.

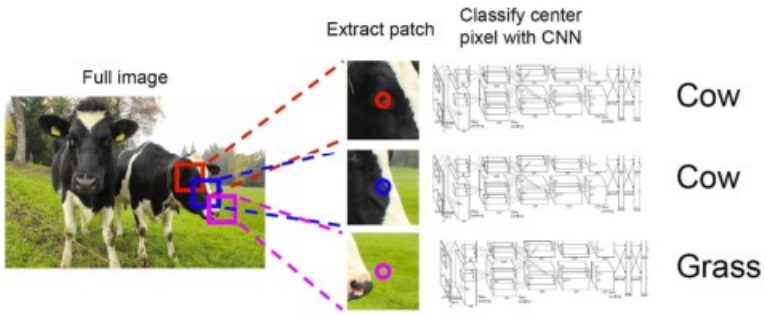


Figure 90: Sliding window for semantic segmentation

Clearly, this approach suffers from different problems:

- First of all, it is very **inefficient**;
- Secondly, it does not exploit the contextual information, i.e. shared features between overlapping patches are not reused.

Second idea: fully convolutional In this case the idea is to exploit the feature extractor part of the AlexNet architecture, in order to use the features to produce the output of each pixel.
As we can see from the image, the network is composed by several **convolutional layers** that are applied to the **input image**, and the result is, for each pixel, a vector of elements representing a **probability distribution**. Despite being **more efficient** than the previous method (in this case we can compute the predictions for the pixels all at once), this method suffers from a **problem**: since we do not use **pooling**, convolutions at original image resolution will be very **expensive**. For this reason, we can consider a variation of this method, which exploits the **encoder-decoder** architecture.

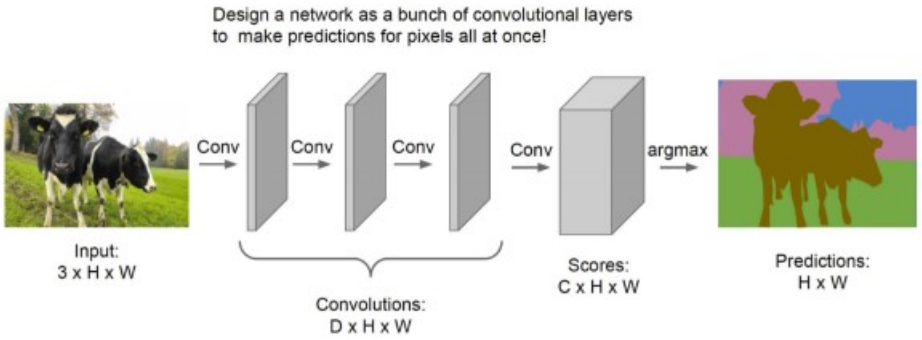


Figure 91: Fully convolutional NN for semantic segmentation

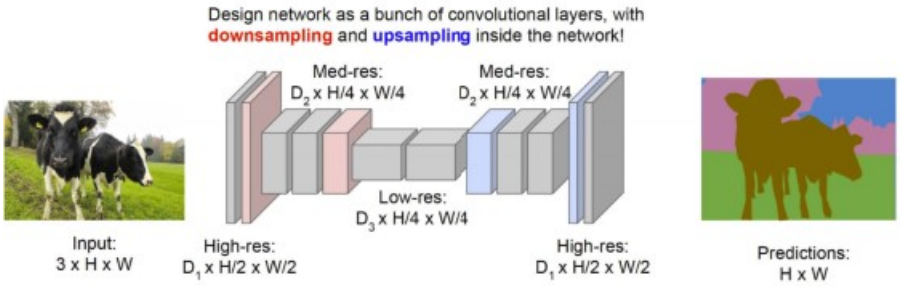


Figure 92: Fully convolutional NN for semantic segmentation with encoder-decoder

Third idea: fully convolutional with downsampling and upsampling As we can see, the network is composed by several **convolutional layers** that are applied to the **input image**, with the difference that in this case **downsampling** and **upsampling** are contained in the network. The downsampling allows to encode the image into a feature map whose dimensionality is lower than the original image, while the upsampling takes as input the output of the downsampling phase, and it produces as output an image with the same size of the original one. The downsampling phase is performed by the **encoder**, while the upsampling is performed by the **decoder**. While for the downsampling phase, an idea could be to exploit the convolutional layers of AlexNet or any other CNN, for the upsampling phase (also called unpooling) the reasoning is not as simple.
In-network upsampling Picture 5.10.1 shows some possible implementations of the unpooling phase:

- *Nearest neighbor*, i.e. the new entries are filled with the same value of the original patch;
- *"Bed of nails"*, i.e. the new entries are filled with 0s;
- *Max unpooling*, i.e. the new entries are filled in the positions corresponding to the maximum elements in the Max pooling layer.

Moreover, we can exploit the same reasoning of pooling to perform the upsampling, in order to obtain the so-called **transpose convolution**, represented in Picture 5.10.1.
However, in 2015 it was discovered that the pipeline did now work well, in particular in solving both the WHAT and WHERE problem:

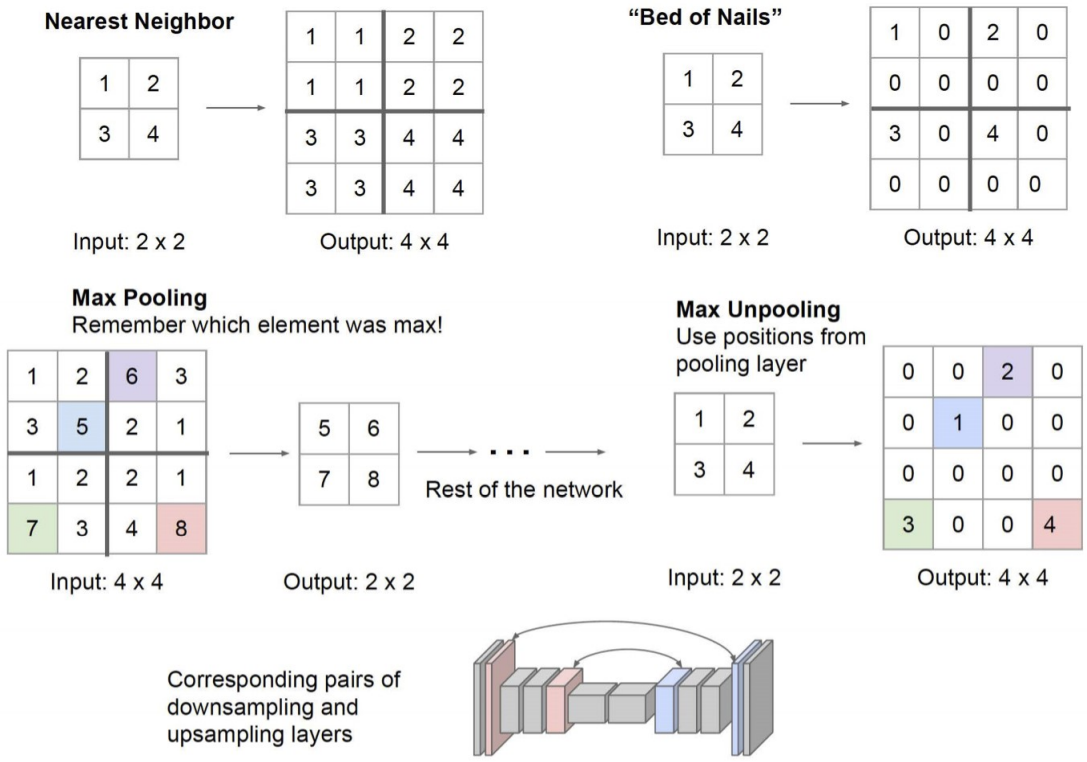


Figure 93: Fully convolutional NN for semantic segmentation with encoder-decoder

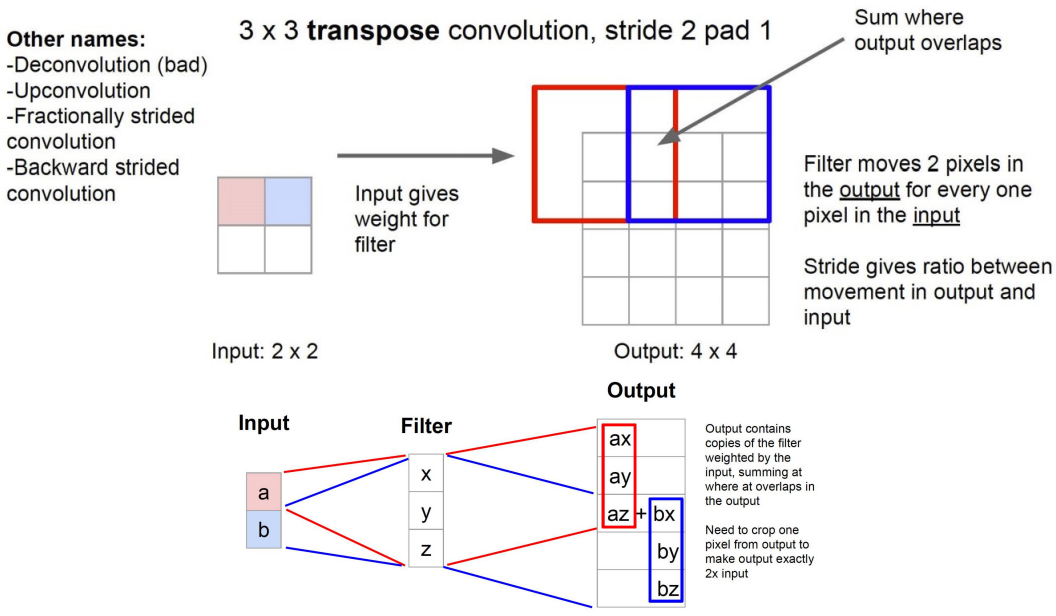


Figure 94: Transpose convolution

- The *what* problem refers to the problem of classifying the prominent object;
- The *where* problem refers to the problem of locating where the object is.

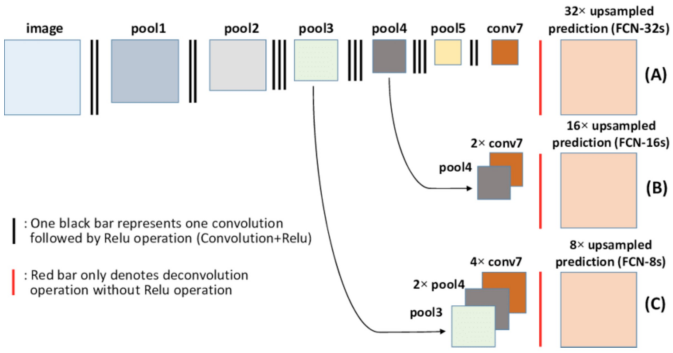


Figure 95: Transpose convolution

If we consider the above Picture, we notice that after layer 7, we extracted powerful features for the WHAT problem, but at the same time we lost some information about the location, i.e. the WHERE problem. In this sense, the idea is to consider an architecture with **branches**, which allow to skip some layers and results in having better performances for the WHERE problem. Picture 5.10.1 compares the results obtained by the original CNN and the ones exploiting the techinque we just described.

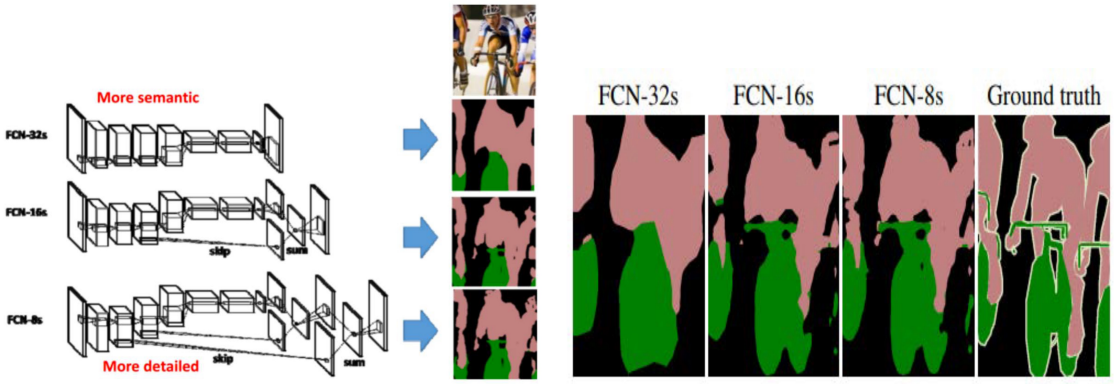


Figure 96: CNN with skipping layers technique

U-net The architecture of the U-net is very similar to the one showed in 5.10.1, with the difference that in this case the network is implemented for medical diagnosis.

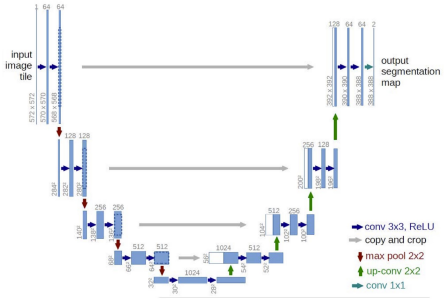


Figure 97: U-net

The evaluation of the network is splitted into:

- A *contraction phase*, which reduces the spatial dimensions, thus increasing the WHAT;
- An *expansion phase*, which recovers object details and dimensions, thus increasing the WHERE.

5.10.2 Object detection

The problem of **object detection** consists of **locating** and **classifying** the object(s) that are represented in an image. In this case we make the following assumptions:

- The image containing a **single prominent object**;
- The object is not rotated, i.e. it is **vertically oriented**.

The idea for solving this problem is the following: we exploit AlexNet (excluding the output layer) to obtain the features of the input image, and then we create two branches:

1. The first one is responsible for solving the **classification** problem;
2. The second one is responsible for solving the **regression** problem, i.e. of predicting the coordinates of the bounding box for the prominent object.

The architecture of the network is represented in 5.10.2.

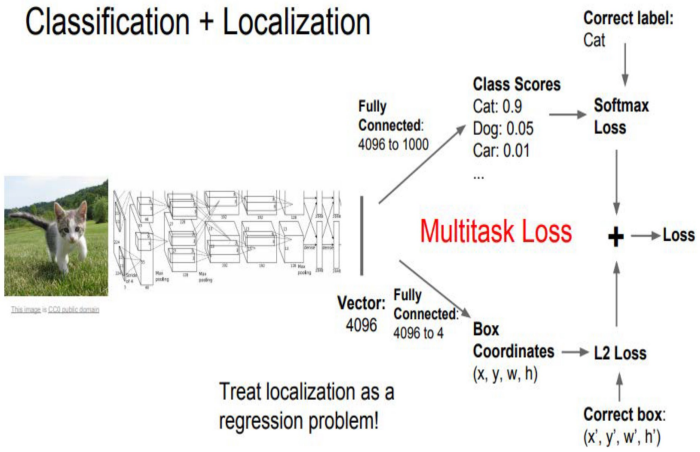


Figure 98: Architecture of the network for solving the object detection problem

As we can see, the two branches are characterized by **two different losses**: the **softmax** is used for the classification branch (each label is characterized by a probability), while the **L2** is used for the regression one. For this reason, the idea is to combine the two losses into a new one, called **multitask loss**, which will be then considered by the backpropagation algorithm for updating the weights of the network.
Now, what can we do when there are more than 1 prominent object represented in an image? The intuition is to create a **new branch** for **each** of the **object**, but clearly this

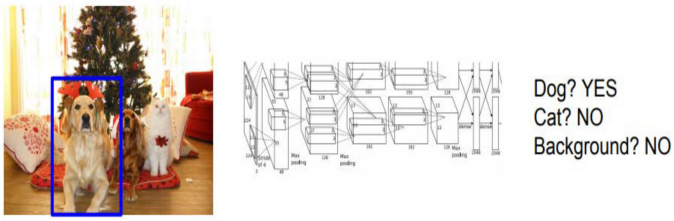


Figure 99: Object detection problem with sliding window approach

number is not known in advance, and for this reason a new approach must be considered. In this case the idea is to exploit a **sliding window** approach, i.e. to apply a CNN to many different crops of the image, in order to classify each crop as object or background. Clearly, the **downside** of this approach is that we need to apply a CNN to a huge number of locations, scales and aspect ratios, which results in having a very computationally expensive operation!
Another idea is to consider the so-called **region proposals**: we find some image regions that are likely to contain objects, and we run the classifier only over that regions. In this way the computation time becomes much lower (e.g. SelectiveSearch gives 2,00 region proposals in a few seconds on CPU).

R-CNN The **R-CNN** is a special CNN which is exploited for solving the object detection (with multiple objects) problem. The architecture is showed in Picture 5.10.2.

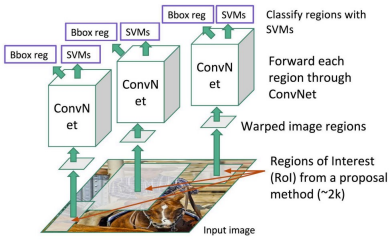


Figure 100: Architecture of R-CNN

Some notes:

- The **input** is represented by an image;
- A proposal method is exploited for finding the regions of interest (RoI, about 2,000) of the image;
- Since each of the RoI may be of different size, they're warped into regions of the same size;
- Each of the warped regions is provided as input to a CNN (excluded of the classification layer), in order to extract the features;
- The output of each of the CNN is provided as input to two different models:
  - An SVM for classifying the region;

- A regression model for estimating the position of the bounding box.

Among the **weaknesses** of this model, we have that:

- The architecture is characterized by a lot of **loss functions**: a loss for the network (*log loss*), a loss for the linear SVM (*hinge loss*) and a loss for the regression model (*least squares*);
- The training phase is extremely slow (84h), and it takes a lot of disk space;
- The detection phase is slow too (47s per image with VGG16);

Notice that both the training and detection time are affected by the fact that a CNN is used for each of the RoI (about 2,000 CNNs).

Fast R-CNN Due to the weaknesses described above, in the following year (2015) a variation of R-CNN was developed, called **Fast R-CNN**, whose architecture is represented in 5.10.2.

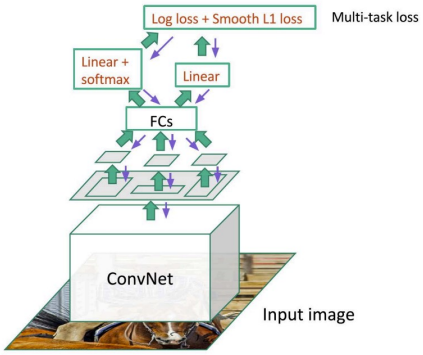


Figure 101: Architecture of Fast R-CNN

Some notes:

- The network takes as **input** an **entire image** and a set of object proposals;
- The networks first processes the whole image with several **convolutional** and **max pooling** layers to produce a convolutional **feature map**;
- For each object proposal, a **RoI pooling layer** extracts a fixed-length **feature vector** from the feature map;
- Each feature vector is fed into a sequence of **fully connected layers** that finally branch into **two** sibling **output layers**:
  - One that produces a **SoftMax** probability estimates over k object classes plus a catch-all *background* class;
  - Another layer that outputs 4 **real-valued numbers** for each of the k object classes.

- The two losses (*log loss* and *smooth L1 loss*) are combined into a *multi-task loss*, which is then used in the backpropagation algorithm for updating the weights of the network.

Picture 5.10.2 provides a comparison between the training and testing time of R-CNN and Fast R-CNN: as we can see, **Fast R-CNN** is **faster** in both training and testing. Moreover, we can notice how Fast R-CNN is characterized by a fully trainable pipeline.

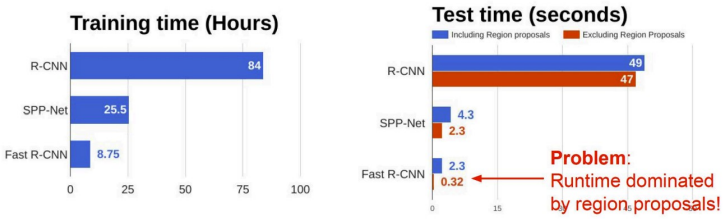


Figure 102: Training and testing time comparison

However, if we consider the **testing time** of Fast R-CNN we notice how it is dominated by the algorithm for searching the region proposal (about 2s on a total of 2.3 s).

Faster R-CNN The last version of the model was proposed in 2016, and in this case the idea is to insert a Region Proposal Network (RPN) in order to predict the proposals from the features extracted by the CNN.

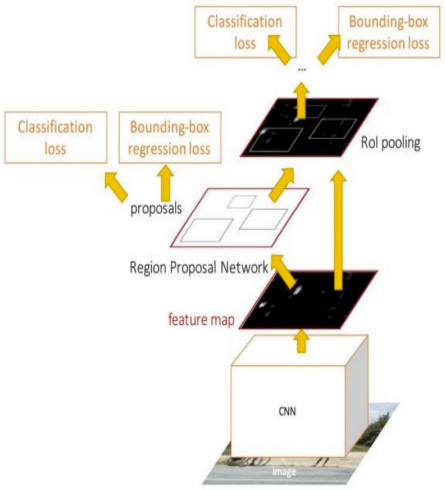


Figure 103: Architecture of Faster R-CNN

In this case, the network is jointly trained with 4 losses:

- The RPN loss for classifying object/non-object;
- The RPN loss for predicting the box coordinates;
- The final classification loss;

- The final bounding box regression loss.

As we can see from Picture 5.10.2, Faster R-CNN represents the model with the lowest test-time, with the peculiarity of not being affected by the time of providing the proposals.

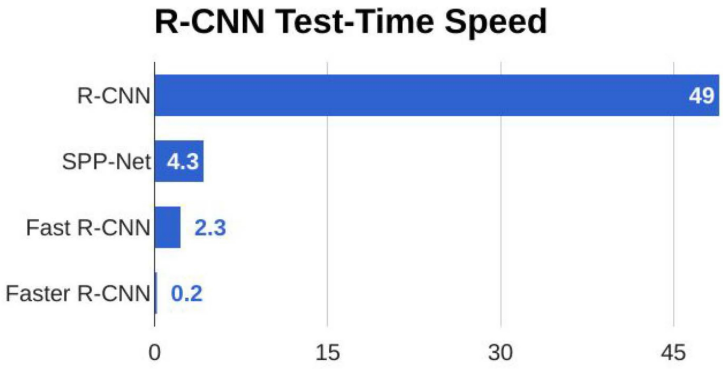


Figure 104: Running times comparison

Finally, notice that also Faster R-CNN represents a fully trainable pipeline.
In general, the arrival of **Deep Learning** in 2012, the performances of object detection algorithms, together with other algorithms for other problems (e.g. speech recognition, NLP etc..) increased significantly.

5.10.3 Human pose estimation

This problem is an instance of the previous one, since the goal is to **detect humans**, but in this case using the **relative positions** of the parts of their **bodies**. Again, the assumption is that the image contains a single prominent human body.
The idea for solving this problem is to represent a **human pose** as a set of 14 joint positions, in order to represent a body with a tree. An example of these positions is given in Picture 5.10.3.

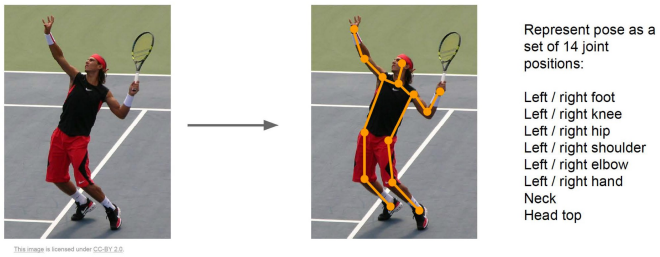


Figure 105: Representation of the joint positions

Now, the intuition is to consider once again the convolutional layers of AlexNet, and to create a branch for each of the 14 positions: each of the networks will then solve a regression problem for estimating the position of each of the parts of the human body. The architecture is showed in 5.10.3.
As in the previous problem, we have a loss for each of the branches (in this case the same loss, L2), which are eventually combined into a unique loss.

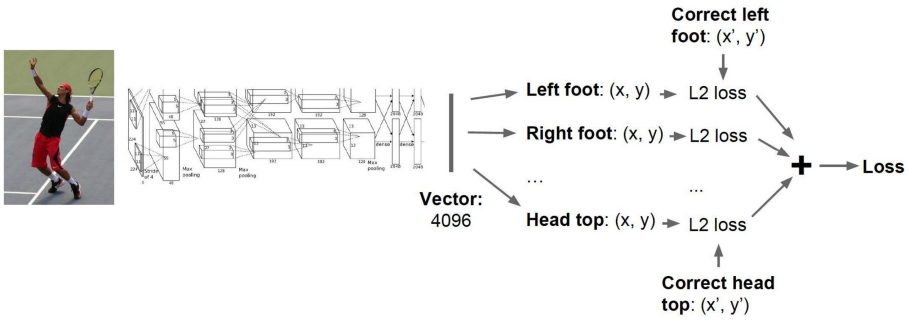


Figure 106: Architecture of the network for solving the human pose estimation problem

5.10.4 Image captioning

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**.

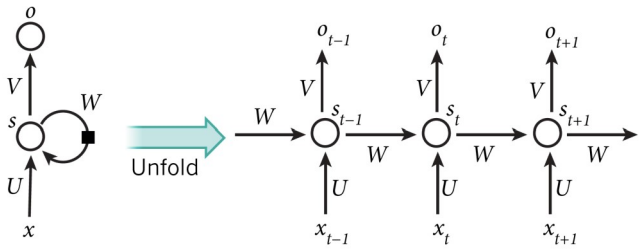


Figure 107: Recurrent neural networks

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_{xh} 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.

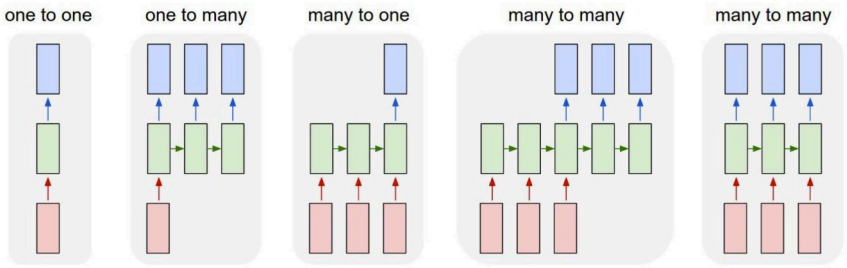


Figure 108: Recurrent neural networks

- 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**.

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.

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

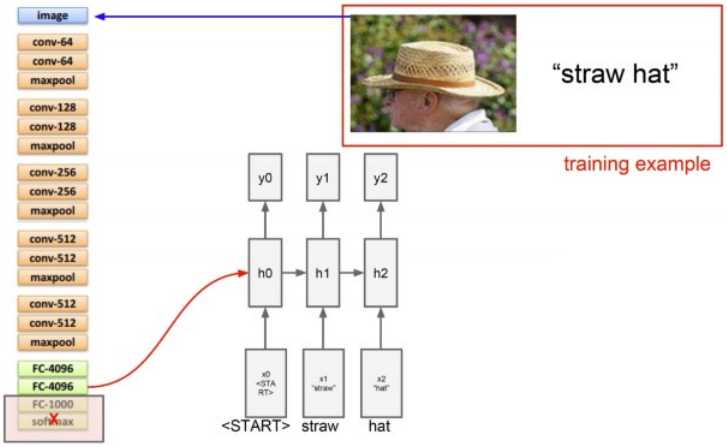


Figure 109: Image captioning: training

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.

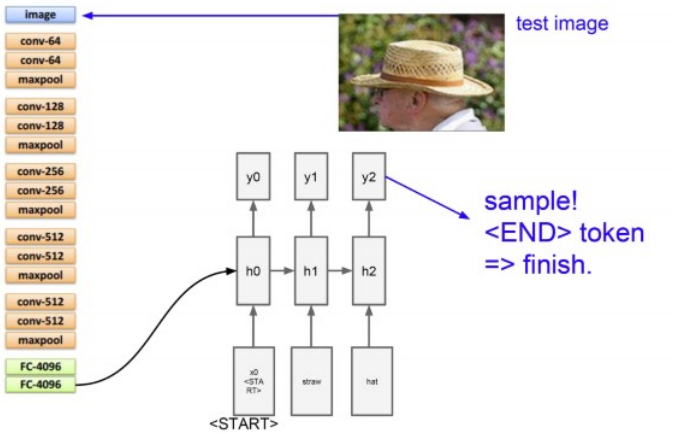


Figure 110: Image captioning: test

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.

6 Dominant-set clustering

We now focus on **unsupervised learning** problems, i.e. **clustering**, and in particular we focus on the **dominant set** approach for solving such problems.

6.1 Graph-theoretic definition of a cluster

These notes are based on [Pavan and Pelillo(2006)].
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.

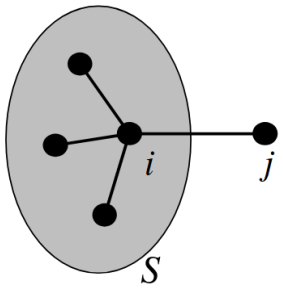


Figure 111: Visual representstion of ϕ

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:

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

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

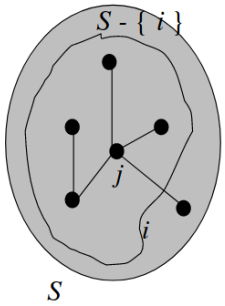


Figure 112: Weight of i w.r.t. the elements in S.

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.

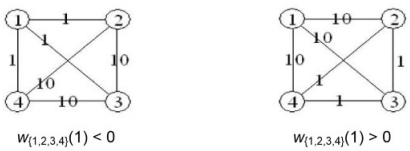


Figure 113: Examples of total weight

As we can see, in the first example we should not add node 1 to the cluster {2,3,4}, 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:

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

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.

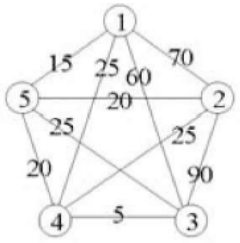


Figure 114: The set of vertices {1,2,3} is dominant.

6.2 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**.

6.2.1 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**:

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

where

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

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

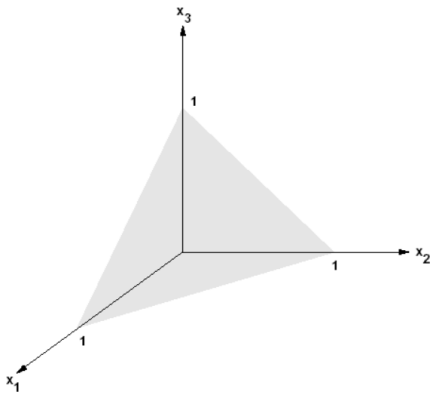


Figure 115: Standard simplex on $\mathbb{R}^3$.

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**, i.e., given a dominant set we can build $x \in \Delta$, which is a strict local maximum of $x^T A x$.

6.2.2 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 **dominant sets** are in **one-to-one correspondence** to **maximal cliques** of G.

6.3 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, and allow to find DS's. The computation is the following:

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

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

Figure 116: MATLAB implementation of discrete-time replicator dynamics

As we can see, the algorithm generates a sequence of points in the standard simplex Δ, and if A is a symmetric matrix, it converges to a strict local maximum.
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.

6.4 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**.

6.5 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 117: Pseudo-code for image segmentation.

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

6.6 Conversational groups detection

[1]

We now focus on the problem of **conversational groups detection**. Formally, a conversational group, also called **F-formations**, exists when *"two or more individuals in close proximity orient their bodies in such a way that each of them has an easy, direct and equal access to every other participant's transactional segment"*. Picture 6.6 provides an example of F-formations.

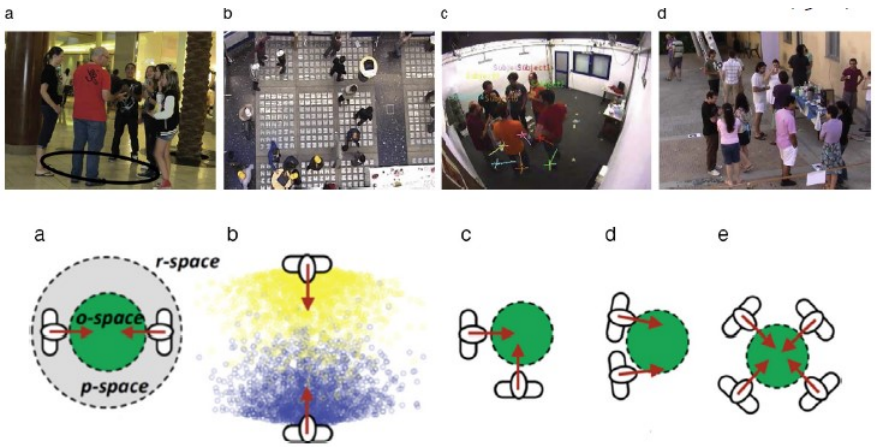


Figure 118: Example of F-formations

Picture 6.6 shows the architecture for solving this problem.
As we can see, the architecture is composed of 4 phases:

[1]This section is based on [Vascon et al.(2016)Vascon, Mequanint, Cristani, Hung, Pelillo, and Murino].

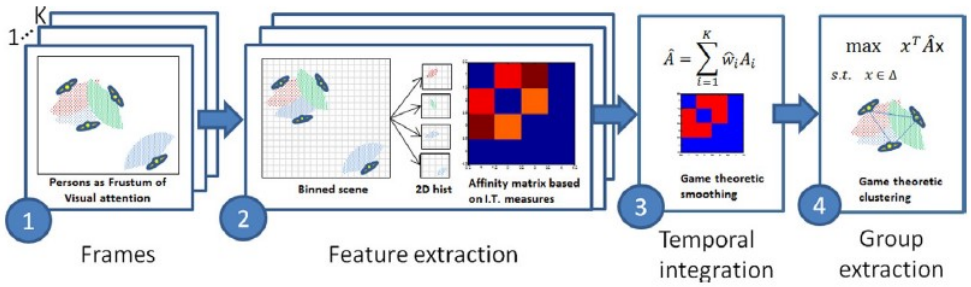


Figure 119: Architecture for solving the groups detection problem

1. The first takes as input an image, and it is responsible for creating the **frames**. In the frames the **persons** are represented with the so-called **frustrum of visual attention**: each **person** in the scene is described by his/her **position** (x, y) and by the **head rotation** θ. The **frustrum** represents the **area** in which a person can sustain a conversation, and it is defined by an **aperture** and by a **length**. The intuition is that the **more** two frustra **overlap**, the **higher** the **probability** of two people to **interact**;
2. The second phase implements the **feature extraction** from the frame: the idea is to create a **graph** in which the **nodes** are the **persons**, while the **edges** represents the **probability** of interaction between them, calculated as the overlap of the corresponding frustra;
3. **Temporal integration**;
4. **Group extraction**, by using the DS approach. In this case, DS allows to execute the extraction without knowing the number of clusters in advance, and to deal very well with the outliers.

More specifically, this model was executed by providing different ways of computing the overlap between the frustra (e.g. K-L divergence, Jaccard etc..), but in all the cases the algorithm performed better than Spectral Clustering (executed providing the correct number of clusters in advance).
Picture 6.6 shows an example of the results: the yellow represents the ground truth, the green represents the result of the model, while red represents the result of another model (HFF).

6.7 Constrained image segmentation

The classical problem of **image segmentation** consists of **partitioning** an **image** into multiple parts or **regions**, often based on the characteristics of the pixels in the image. Modern **variations** of the classical (purely bottom-up) approach, involve, e.g., some form of user assistance (**interactive segmentation**) or ask for the simultaneous segmentation of two or more images (**co-segmentation**). At an abstract level, all these variants can be thought of as **"constrained" versions** of the original formulation, whereby the segmentation process is guided by some external source of information.
In the following sections we describe a model for solving the interactive image segmentation problem.

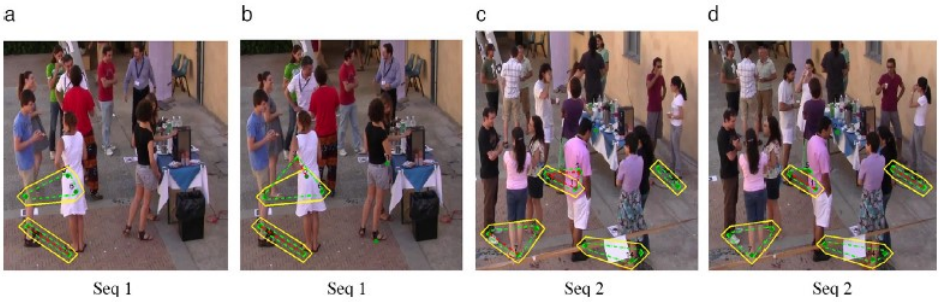


Figure 120: Results of the model

6.7.1 Constrained dominant sets

Before analyzing the model, we briefly introduce the **constrained DS approach**. The setting is the following: given $S \subseteq V$ and a parameter $\alpha > 0$, we define the following parameterized family of quadratic programs:

$$\begin{array}{ll} \max & f_s^{\alpha}(x) = x^T(A - \alpha \hat{I}_S)x \\ \text{s.t.} & x \in \Delta \end{array} \tag{1}$$

, where $\hat{I}_S$ is the **diagonal matrix** whose elements are set to 1 in correspondence to the vertices outside S, and to 0 otherwise. In this way, the elements of A in the diagonal will be set to $-\alpha$ in correspondence to the vertices outside S, and to 0 otherwise, or in other words:

$$a_{ii} = \begin{cases} -\alpha & \text{if } i \notin S \\ 0 & \text{otherwise} \end{cases}$$

An important **property** of this formulation is that by setting

$$\alpha > \lambda_{\text{MAX}}(A_{V-S})$$

, all the **local solutions** of 1 will have a **support** containing elements of S. Notice that $\lambda_{\text{MAX}}(A_{V-S})$ represents the **largest eigenvalue** of the sub-matrix of A containing the vertices outside S.
In general, solving 1 allows to find the **dominant sets that contain the set of nodes** $S \subseteq V$.

6.7.2 Interactive image segmentation [2]

The **interactive image segmentation** represents a variant of the classical image segmentation problem. In this case the **input** of the model is given by an **image** and some **annotations**, indicating the region of interest in which performing the segmentation, while the **output** contains the **segmentation**. There exit two types of possible annotations:

- **scribbles**, representing the **foreground** and the **background**;

[2] This section is based on [Zemene et al.(2018b)Zemene, Alemu, and Pelillo].

- **bounding box**, which allows the **selection** of a portion of the image.

Picture 6.7.2 provides an example of the two possible annotations.

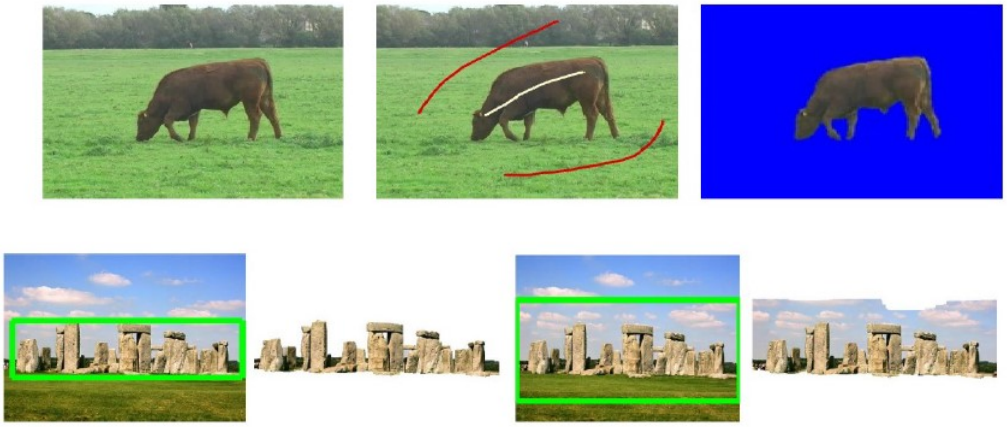


Figure 121: Types of annotations

Despite being very **different** from each other, we underline the fact that the **CDS approach** allows to **solve** the interactive image segmentation problem with **both** types, as represented in 6.7.2.

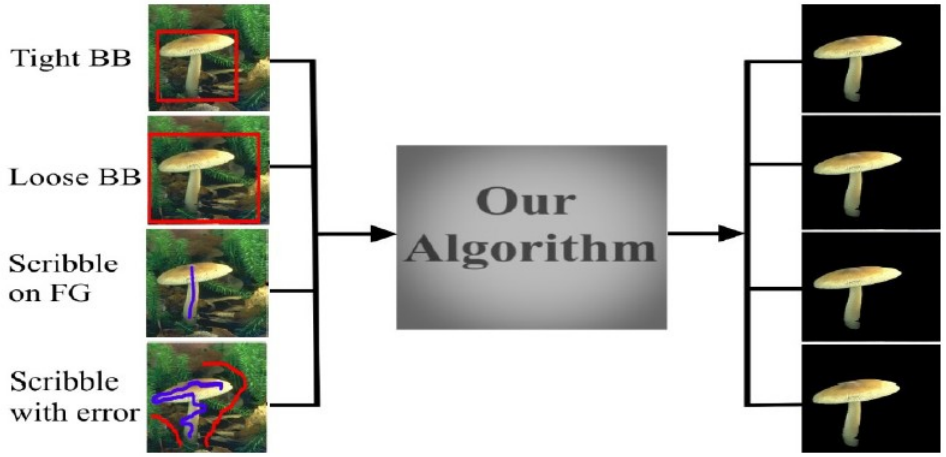


Figure 122: Dominant set approach for IIS

The **framework** for solving this problem is showed in 6.7.2.
The steps are the following:

1. We first extract the so-called **super pixels**, i.e. small contiguous regions of pixels in which the color is homogeneous. This steps allows to reduce the **dimensionality** of the problem;
2. Then, the idea is to create a **graph**, where each of the super pixel is a node. Moreover, we define a partition of this graph: the set S, containing the pixels

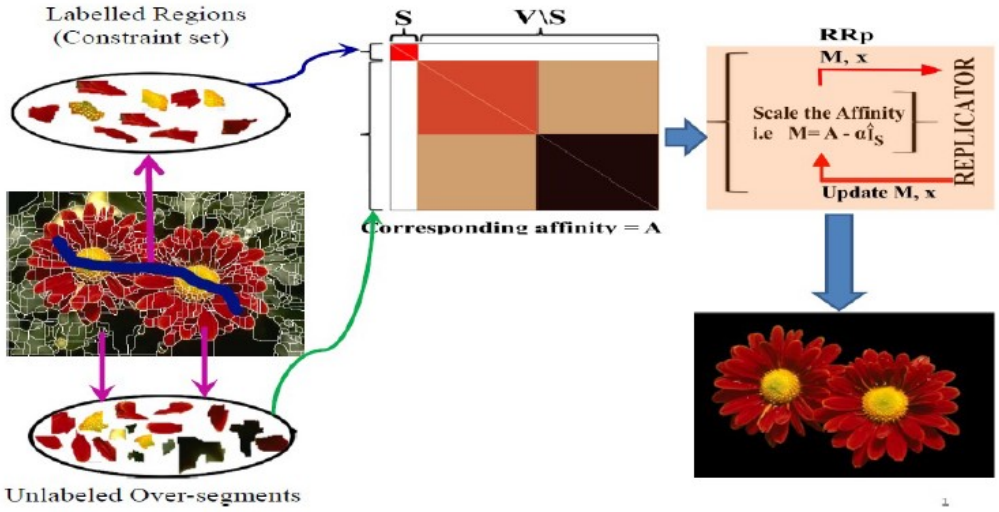


Figure 123: Framework

touched by the scribble, and the set $S - V$ containing all the other pixels. We define the corresponding **affinity matrix**;

3. We run several times the **replicator dynamics** algorithm, with input x and the scaled matrix $M = A - \alpha \hat{I}_S$, and we get the dominant sets. Let L be number of extracted dominant sets, and say their union is UDS $= D_1 \cup D_2 \cup .. \cup D_L$, then:

   - For scribble-based approach, UDS represents the segmentation result;
   - For boundary-based approach, its complement V-UDS represents the result. A bounding box can be indeed consider as a square of scribble, so we need to take the inverse of the union, since the union would retrieve the segmentation of the background.

Picture 6.7.2 shows some results of the model. As we can see, testing both the scribble and the bounding box results in having a segmentation which is highly similar to the ground truth.

6.8 Large-scale image geo-localization

This section is based on [Zemene et al.(2018a)Zemene, Tesfaye, Idrees, Prati, Pelillo, and Shah]. The **goal** of this problem is to provide the **location**, in terms of **longitude** and **latitude**, of the location represented in the input image.
In this case the **dataset** is composed of **images** and the relative **coordinates**, and the goal of the model is to compare the input image with the dataset, and return the coordinates of the closest image. More specifically, for each location the dataset is composed of:

- A set of **4 side views** plus **1 top view**;
- A **label**, corresponding to the **name** of the location, the **longitude** and the **latitude**.

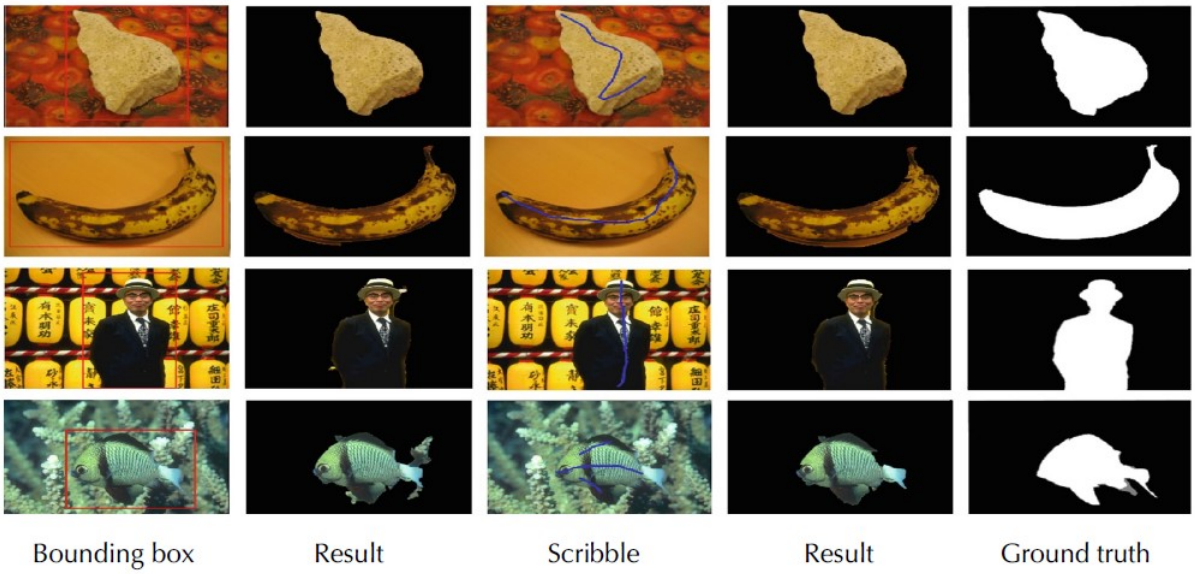


Figure 124: Results

Figure 125: Image geo-localization

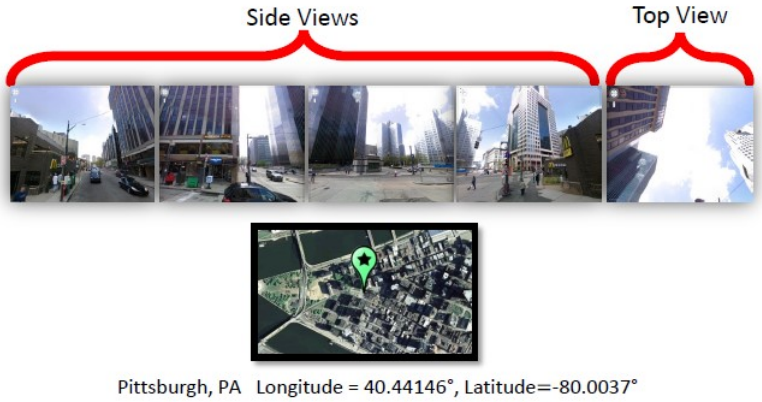


Figure 126: Dataset

An example of the dataset is provided in Picture 6.8.
The **pipeline** for solving this problem is represented in Picture 6.8.

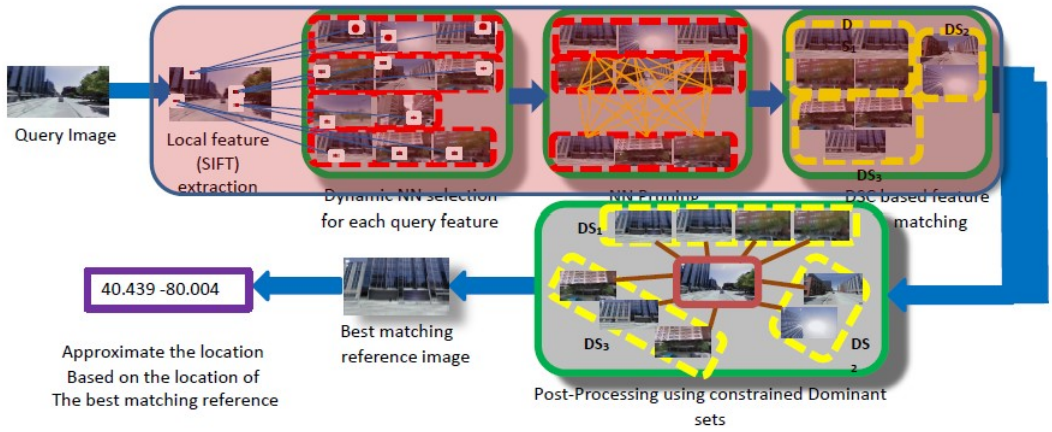


Figure 127: Pipeline

As we can see:

1. The model is provided as **input** with the **query image**, i.e. the image for which the location must be localized;
2. The first operation is the **local feature (SIFT) extraction**. SIFT features (scale-invariant feature transform) are **local** and **based** on the **appearance** of the object at particular interest points, and are invariant to image scale and rotation. They are also **robust** to changes in **illumination**, **noise**, and **minor changes** in viewpoint;
3. Then, the **dynamic NN selection** occurs: each of the SIFT features is used to get a **set of the closest images** (and corresponding features) in the dataset;
4. Then, the **pruning** phase removes portions of the image corresponding to some outliers;
5. Then, DS are used for **clustering** the features, i.e. for matching similar features;
6. **Post-processing** using CDS;
7. Selecting the **best matching reference image**;
8. Approximate the final **location** based on the location of the best matching reference image.

The model was trained and tested with the following datasets:

- The **first dataset** is composed of:
  - A train set composed of 102k Google street view images from Pittsburgh, PA and Orlando, FL;
  - A test set composed of 521 GPS-tagged unconstrained images.
- The **second** one is composed of:

- A train set composed of 300k Google street view images from 14 different cities from Europe, North America and Australia;
- A test set composed of 500 GPS-tagged unconstrained images.

From the **quantitative results** we notice how the **CDS** approach followed by the **post-processing** resulted the best model. Picture 6.8 shows the qualitative results of the model.

Figure 128: Qualitative results

6.9 Multi-target tracking in multiple non-overlapping cameras

This section is based on [Tesfaye et al.(2017)Tesfaye, Zemene, Prati, Pelillo, and Shah]. We now study an application of CDS for solving two related problems:

- **Tracking objects** across several cameras;
- **Person re-identification** in different cameras.

6.9.1 Person re-identification

We start our discussion with the person re-identification problem. As we introduced before, here the goal is to **recognize an individual** over different non-overlapping **cameras**. Thus, given a **gallery** of person images we want to recognize (between all of them), a new observed image, called **probe**.
Notice that in this case we're not interested in knowing the identity of the person, but just in recognizing him/her over different cameras. Moreover, in modern algorithms we do not consider a single frame, but instead we consider a **sequence of consecutive frames**, thus both the probe and the gallery are represented by a sequence of overlapping bounding boxes, as showed in 6.9.1. In this sense, this approach is **video-based**.
For what concerns the **solving method** we distinguish:

Figure 129: Person re-identification

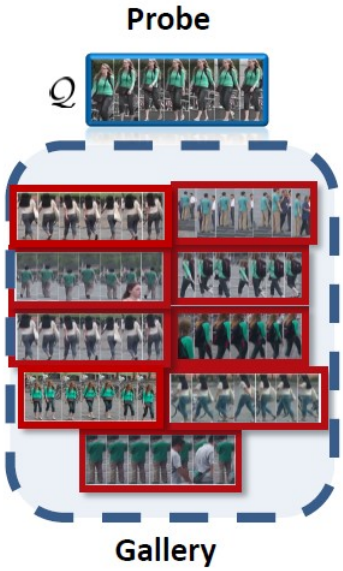


Figure 130: Probe and gallery in video-based person re-identification

- The **traditional methods**, which focus on:
  - Building a better **feature representation** of the objects;
  - Building a better **distance metric**, i.e. considering a good similarity function, in order to consider both appearance information and motion information;
  - Ranking the images of the gallery based on the **pairwise similarity** distances from the query image.
- The **approach of the paper**, which focuses on:

- Using **standard features** and **distance metric**;
- Extracting **DS's** for each query image;
- Performing ranking over **shortlisted clips**, and NOT over the whole set.

In general, the paper takes into account both the **relationship** between **query** and elements in the **gallery** and the relationship between **elements in the gallery**.
More specifically, the goal of the provided model is to find in the **gallery DS's of images that contain the query image Q**, i.e. images that are similar to Q and similar to each other (this explains the usage of constrained DS). The pipeline is showed in Picture 6.9.1.

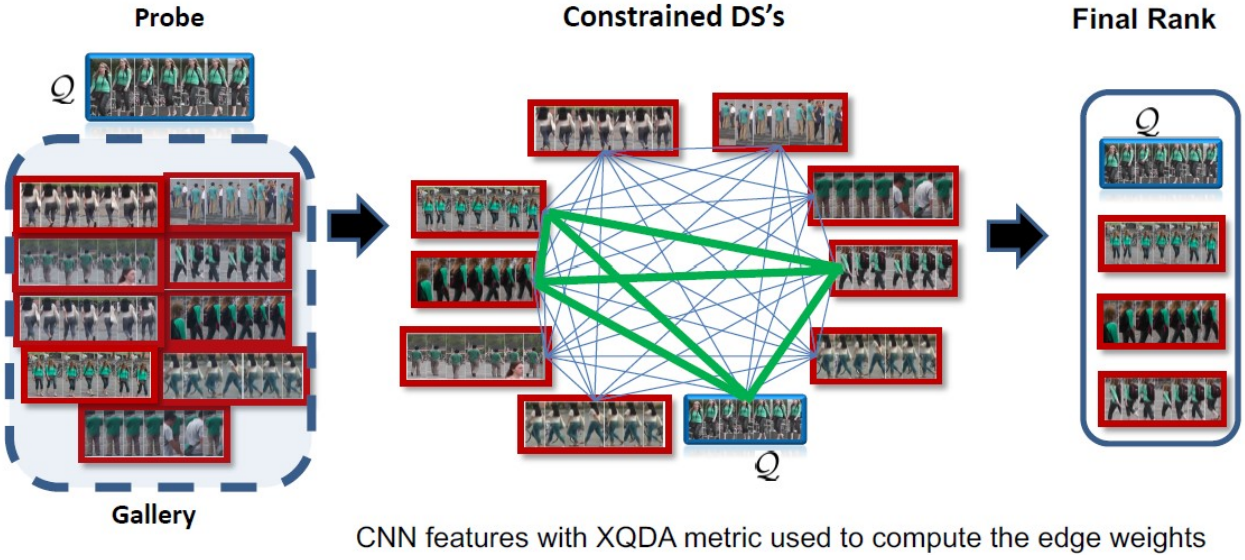


Figure 131: Person re-identification with CDS

As we can see:

1. The gallery images and the probe Q form a **graph**, where each node represents a **sequence of frames**, and the **weighted edges** represent the similarities between frames considering both appearance and motion;
2. CDS is used for retrieving the **clusters** of frames similar to Q and to each other;
3. The frames of the clusters are **ranked** w.r.t. the similarity with Q.

The model was trained and tested with the largest video re-identification dataset, produced by 6 near-synchronized cameras, and composed of 1,261 identities, 3,248 distractors and tracklets of 25-30 frames long. The qualitative results showed how the model outperformed the existing ones; Picture 6.9.1 shows the results: the green and red boxes denote the same and different persons with the probes, respectively. Notice that the gallery images are ordered based on their membership score, from the highest to the lowest.

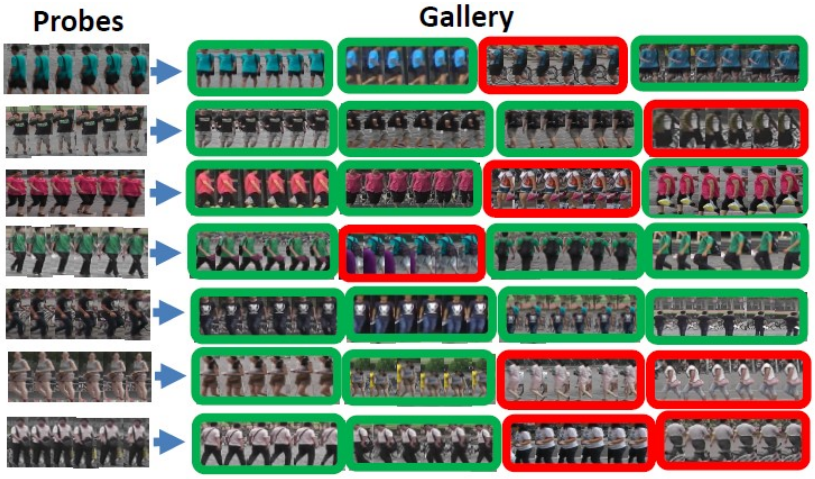


Figure 132: Results

6.9.2 Multi-target multi-camera tracking

We now consider another problem, which is related to the previous one, or rather, the **multi-target multi-camera tracking problem**. This problem consists on **tracking multiple objects from multiple non-overlapping cameras**. In this sense, it can be considered a more general version of the previous problem.

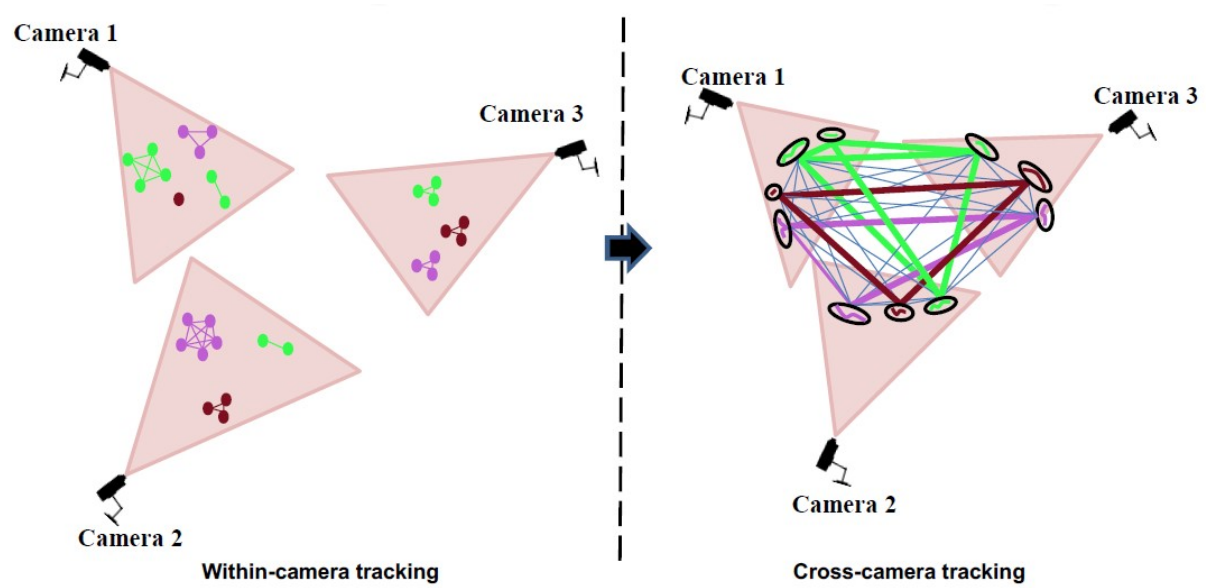


Figure 133: Multi-target multi-camera tracking

The **pipeline** of the model that solves the problem is showed in Picture 6.9.2.
The steps are the following:

1. First of all, an **human detection algorithm** is executed on the frames of the camera, in order to detect the humans;

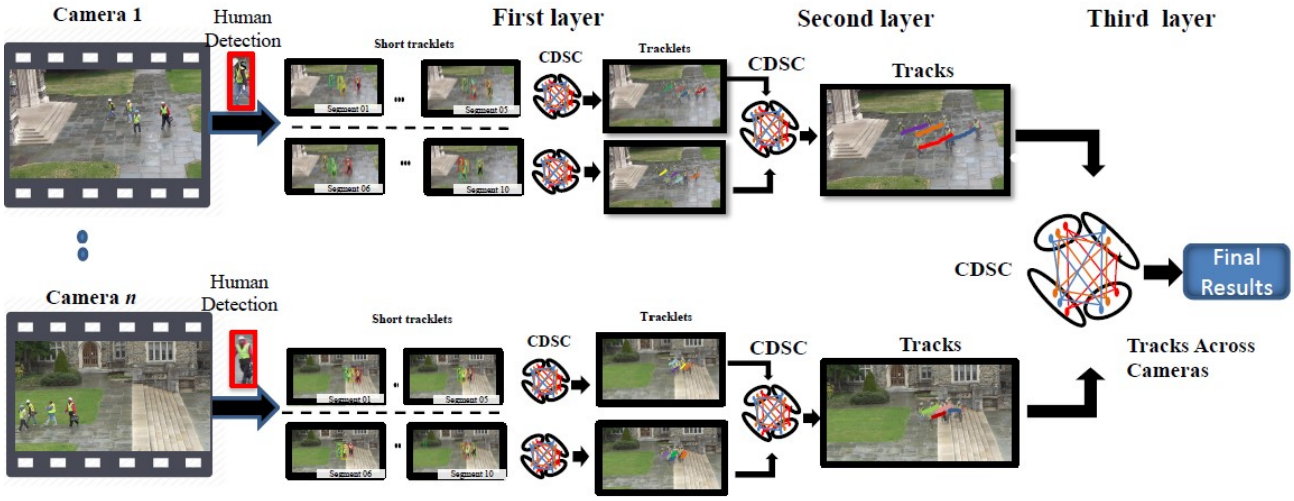


Figure 134: Pipeline

2. First layer: here, the **short tracklets** obtained by the detection algorithm are represented as a **graph**. Notice that a short tracklet represents a sequence of consecutive bounding boxes concerning the same person, and all the tracklets compose the nodes of the graph. On the other hand, the **weights** between the tracklets combine appearance and motion similarity, where:

   - **Appearance similarity** is given by some CNN features;
   - **Motion similarity** is given by constant velocity.

   Then, CDS **clustering** is performed on this graph, obtaining longer tracklets as clusters, as represented in 2.

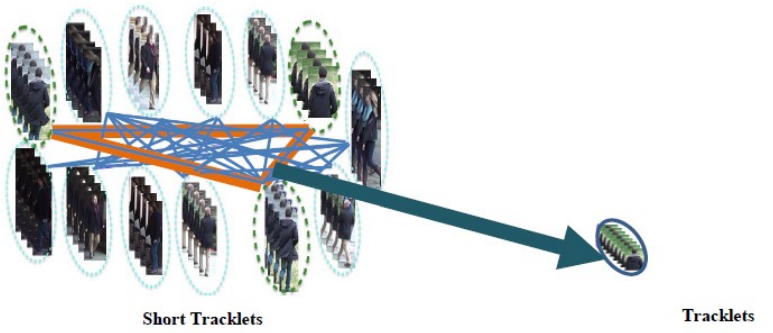


Figure 135: Tracklets extraction

Picture 2 shows in a more detailed level the execution of the first layer.

As we can see, the short tracklets form the graph, where the weight of each edge is given by the displayed formula. Then, the DS clusters are displayed.

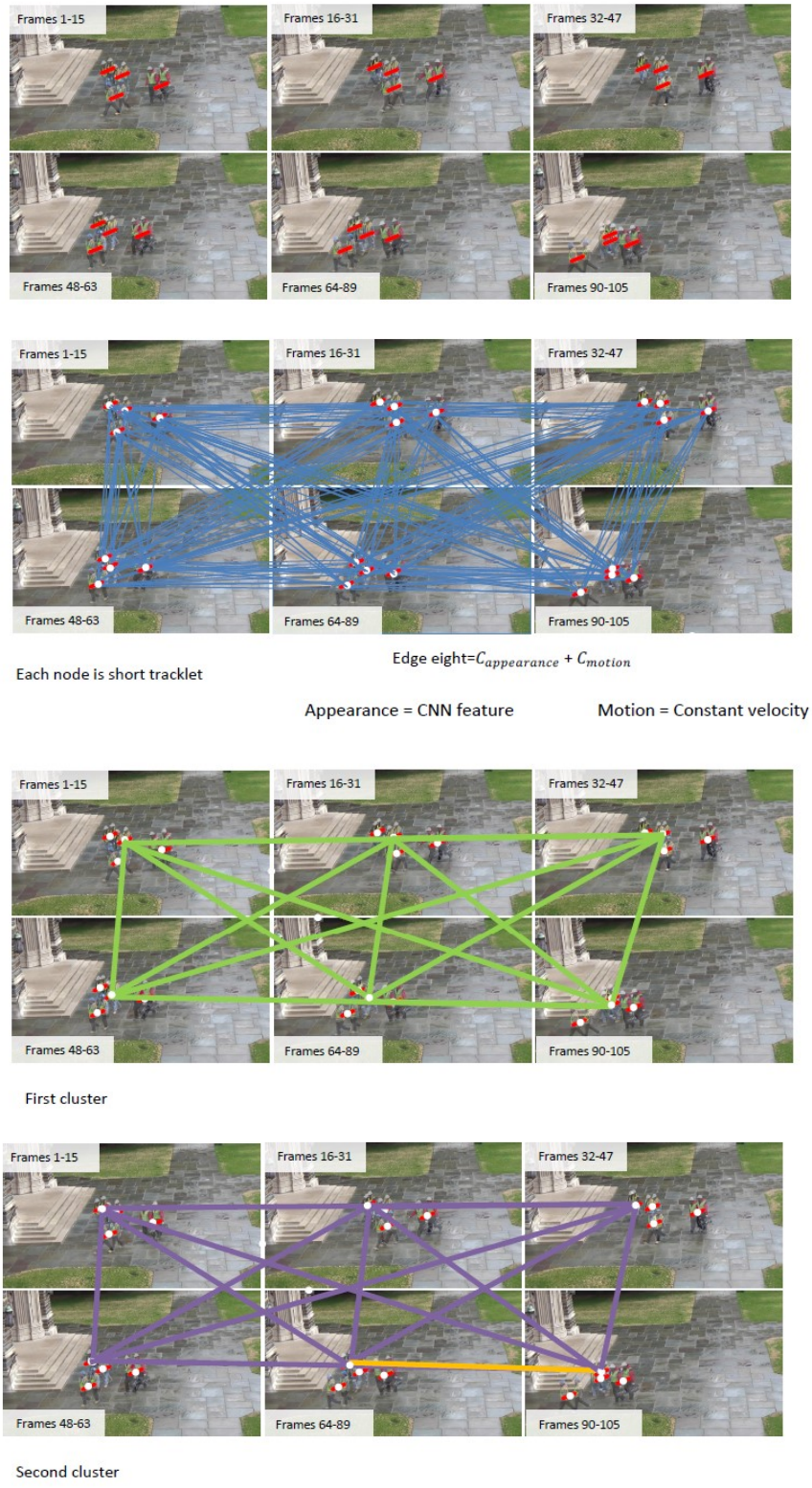


Figure 136: First layer: detail

3. Second layer: here a new **graph** is formed, where the nodes are represented by the tracklets returned by the previous phase, and the edges remain the same. Again,

CDS **clustering** is performed, obtaining **tracks** as clusters, as represented in 3.

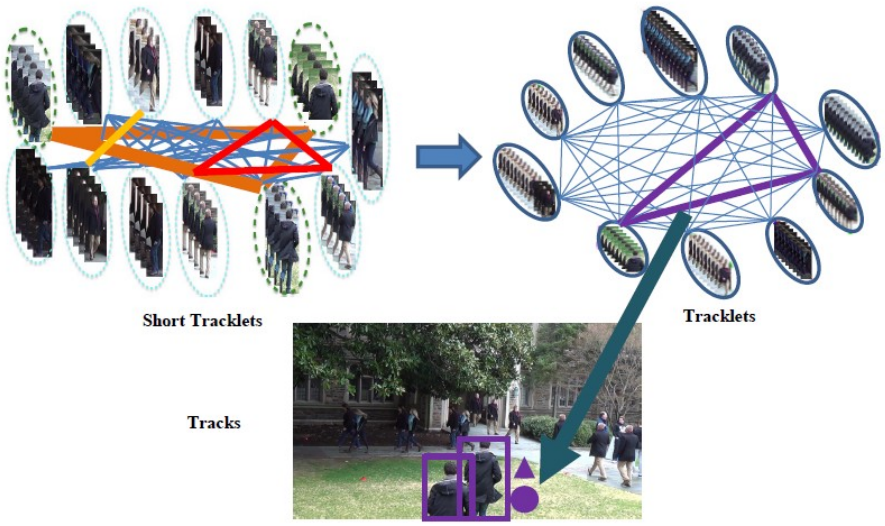


Figure 137: Tracks extraction

4. Third layer: again, a new **graph** is built using the tracks obtained in the previous layer across all the **cameras** as nodes, and again a CDS clustering is performed, using the **cameras** as **constraints**, as showed in Picture 4.

Picture 6.9.2 shows a summary of the functioning and the results of the model.
The model was trained and tested with the largest MTMC dataset, obtained by 8 fixed synchronized cameras and composed of more than 2 million frames, each containing from 0 to 54 persons, and 2,700 identities. The quantitative results show that:

- The fraction of computed detections that are correctly identified (**IDP**) is **higher** than the one of two other models;
- The fraction of ground-truth detections that are correctly identified (**IDR**) is **higher** than the one of two other models;
- The ratio of **correctly identified** detections over the **average number of ground-truth** and **computed detections** is **higher** than the one of two other models;

Finally, Picture 6.9.2 shows the qualitative results obtained by the model.

6.10 Conclusions

In conclusion, **DS** and related concepts shown to be a **powerful** notion for attacking a variety of **CV problems**, and ongoing work focuses on **combining deep learning and DS's for improving performances**.

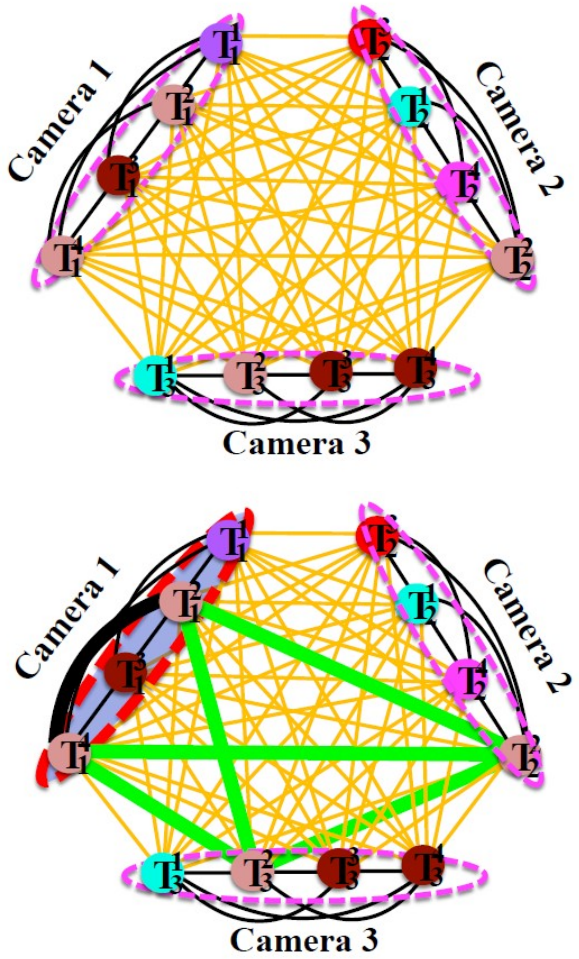


Figure 138: Third layer

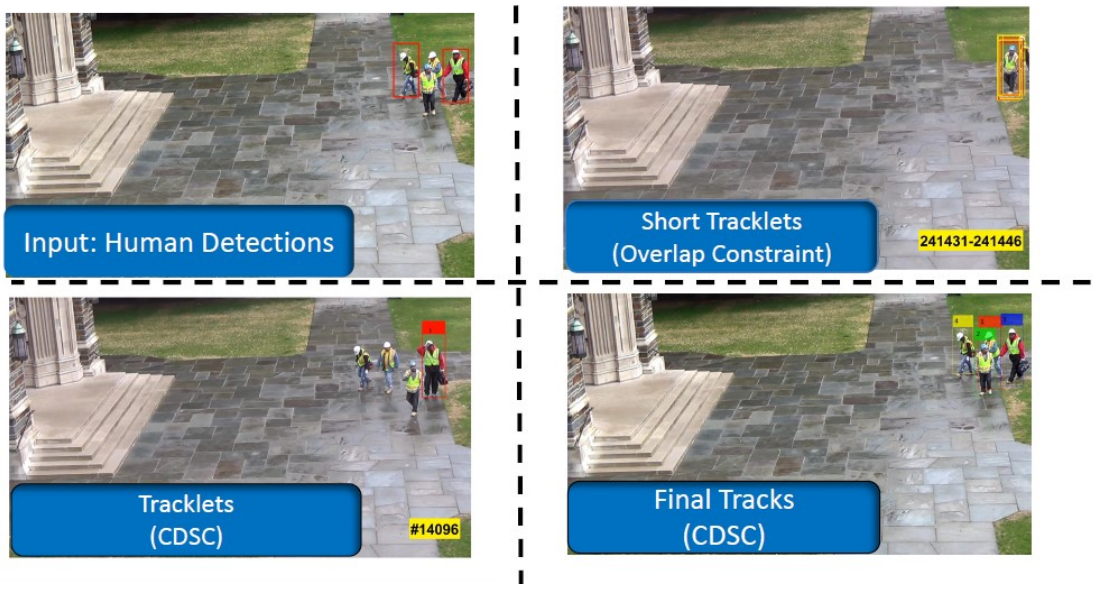


Figure 139: Resume

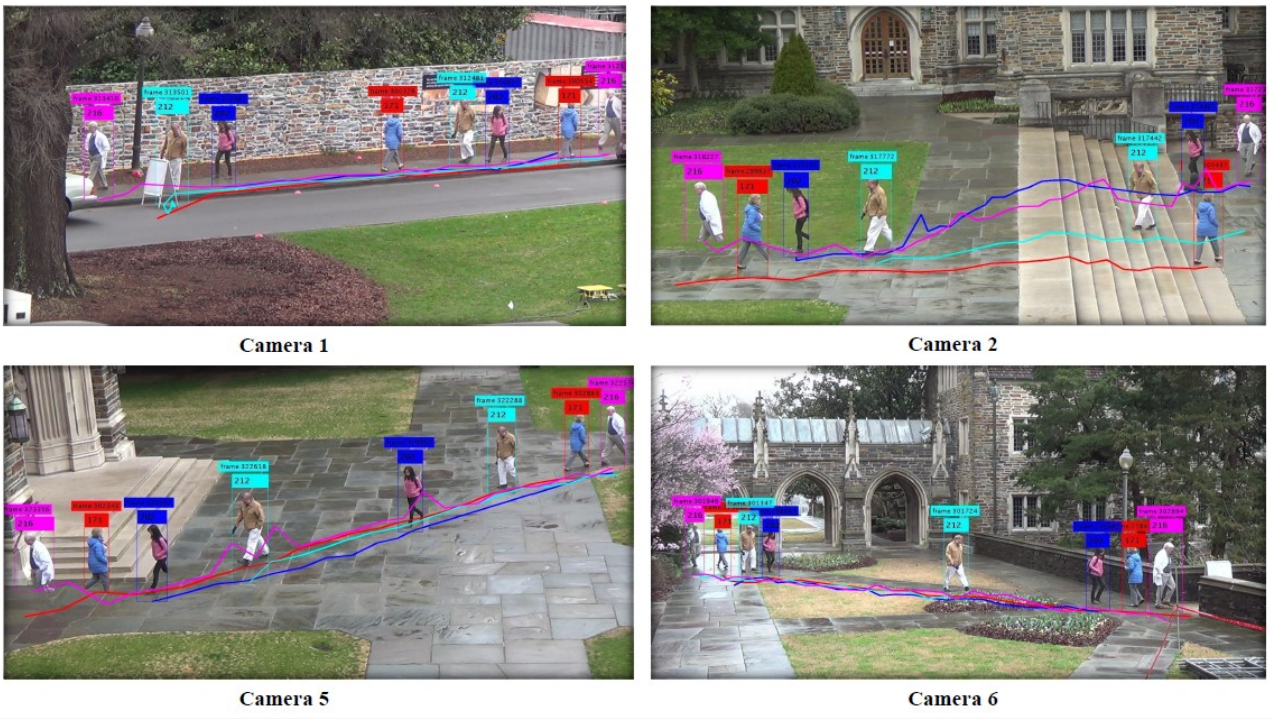


Figure 140: Results

7 Context aware models of classification

The standard approach, showed in Picture 7, when dealing with **pattern recognition** problems is to;

1. **Pre-process** our data;
2. **Extract** some useful **features**;
3. Perform the **classification**.

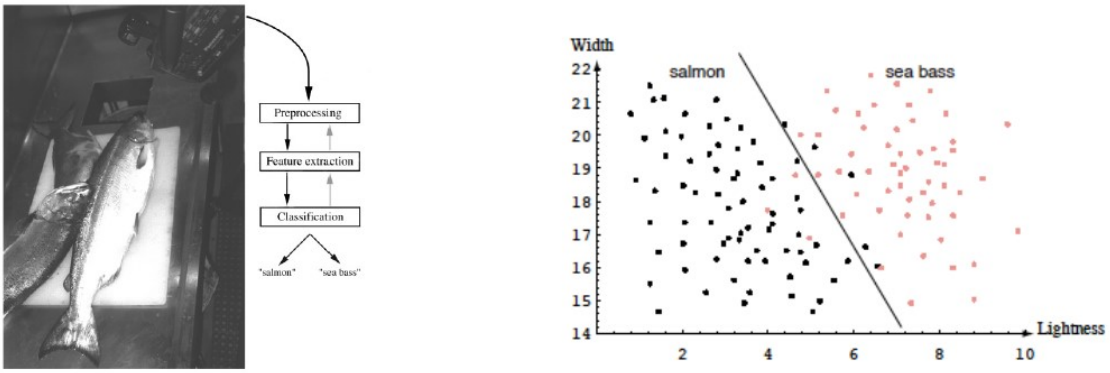


Figure 141: Standard approach

However, there are some situations in which the features are ambiguous, or do not help.

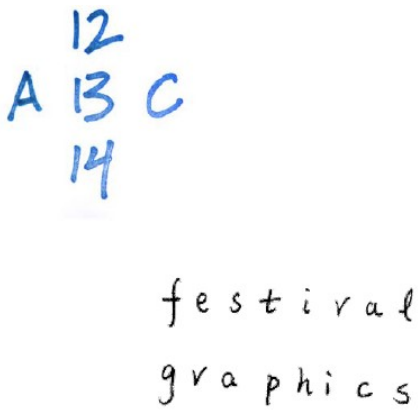


Figure 142: Contextual information

For example, as showed in 7, we need to exploit contextual information in order to classify a digit/letter, or to pronounce a given letter or to infer the char in a sentence.

7.1 The consistent labeling problem

A **labeling problem** involves:

1. A set of n **objects** $B = b_1, b_2, .., b_n$;
2. A set of m **labels** $\Lambda = 1, 2, .., m$

, and the **goal** is to **label each object** of B with a label of Λ. To this end, two sources of information are exploited:

- **Local measurements**, which capture the **salient features** of each object viewed in isolation;
- **Contextual information**, expressed in term of a **real-valued matrix of compatibility coefficients** $R = r_{ij}(\lambda, \mu)$, where $r_{ij}(\lambda, \mu)$ measures the strength of compatibility between the two hypotheses :

  1. b_i is labeled λ;
  2. b_j is labeled μ.

  An important property is that these **compatibility coefficients** can be **learned** from **data**, and the matrix R contains probability distributions.

Initially, the situation is the following:

- $0 \leq p_i(\lambda) \leq 1, \forall \lambda \in \Lambda$;
- $\sum_{\lambda \in \Lambda} p_i(\lambda) = 1$.

, i.e. each object b_i is characterized by a probability ditribution over the space of all the labels.

7.2 Relaxation labeling processes

In a classic 1976 paper, Rosenfeld, Hummel and Zucker introduced the following **update rule** (assuming a non-negative compatibility matrix):

$$p_i^{(t+1)}(\lambda) = \frac{p_i^{(t)}(\lambda) q_i^{(t)}(\lambda)}{\sum_\mu p_i^{(t)}(\mu) q_i^{(t)}(\mu)} \tag{1}$$

, where

$$q_i^{(t)}(\lambda) = \sum_j \sum_\mu r_{ij}(\lambda, \mu) p_j^{(t)}(\mu)$$

quantifies the **support** that **context** gives at time t to the hypothesis "b_i is labeled with label λ".
Notice that in 1, if both $p_i^{(t)}(\lambda)$ and $q_i^{(t)}(\lambda)$ assume **high value**, then $p_i^{(t+1)}(\lambda)$ **increases**, whereas if they assume **small values**, $p_i^{(t+1)}(\lambda)$ **decreases**. Notice that the **denominator** represents a **normalization factor**, since we recall that the matrix is a probability distribution, so the elements must sum up to 1.
This rule is used in order to make the **initial matrix** aware of the **contextual information**, and since their introduction in the mid-70's, relaxation labeling algorithms have found **applications** in virtually all problems in **CV** and **pattern recognition** (e.g. region-based segmentation, graph matching, handwritten interpretation etc..). Moreover, intriguing **similarities** exist between **relaxation labeling processes** and certain mechanisms in the **early stage of biological visual systems**.

7.3 Hummel and Zucker's consistency

In 1983, Hummel and Zucker developed an elegant theory of **consistency** in labeling problem. By analogy with the unambiguous case, which is easily understood, they define a **weighted labeling assignment** p **consistent** if:

$$\sum_{\lambda} p_i(\lambda) q_i(\lambda) \geq \sum_{\lambda} v_i(\lambda) q_i(\lambda) \tag{1}$$

for all labeling assignments v.
This basically represents a **generalization** of the **classical** (Boolean) **constraint satisfaction problems**, since in a general CSP we want to assign the vertices some labels according to some logical constraints. In this case, we have two differences:

1. The **constraints** are **soft**, not logical;
2. The **labeling** we use are not hard, but **probabilistic**. In this sense, each vertex is not assigned a single label, but a probability distribution over all the possible labels.

7.4 Relaxation labeling as a non-cooperative game

As observed by Miller and Zucker (1991) the consistent labeling problem is equivalent to a non-cooperative (polymatrix) game. In such formulation we have:

- Players = Objects;
- Pure strategies = Labels;
- Mixed strategies = Weighted labeling assignments;
- Payoffs = Compatibility coefficients,

and the concept of **Nash equilibrium** can be put in one-to-one correspondence to the one of **consistent labeling**.
Further, the Rosenfeld-Hummel-Zucker update rule corresponds to discrete-time multi-population replicator dynamics.

7.5 Application to semi-supervised learning

Differently from a classical supervised learning, in which all the examples of the dataset are associated with a label, in the case of **semi-supervised learning** only a **fraction** of the dataset is **labeled**, so the goal is to use the labeled objects to classify the unlabeled ones.
In this sense, the final model will take into account both labeled and unlabeled data points.

7.6 Graph transduction

The problem of **graph transduction** can be formulated as follows: given a set of data points grouped into:

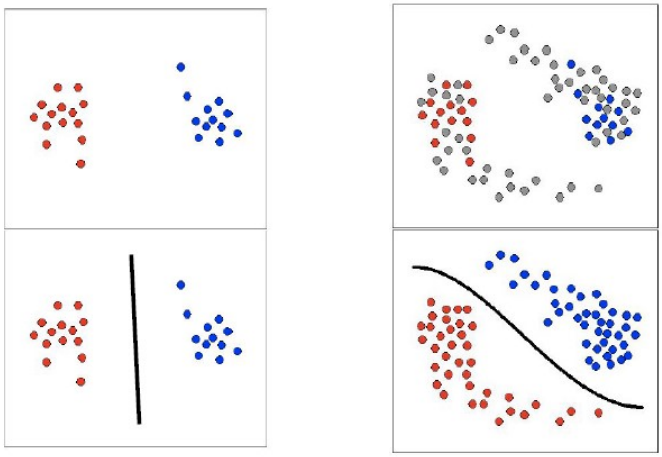


Figure 143: Semi-supervised learning

- **Labeled** data $(x_1, y_1), (x_2, y_2), .., (x_l, y_l)$;
- **Unlabeled** data $x_{l+1}, .., x_n$, with $l \ll n$

, we can express a graph $G = (V, E)$ where:

- V are the **nodes** representing **labeled** and **unlabeled** points;
- E are the **edges** between nodes **weighted** by the similarity between the corresponding pairs of points.

Then, the goal is to **propagate** the information **available** at the **labeled nodes to unlabeled ones** in a "consistent" way. In this case consistent means that the data form **distinct clusters**, and two points in the same cluster are expected to be in the same class. (CSP problem!)
A simple case of graph transduction in which the graph G is an unweighted undirected graph:

- An edge denotes perfect similarity between points;
- The adjacency matrix of G is a $0/1$ matrix.

An important **property** of this problem is that it can be formulated as a **non-cooperative game**. Given a weighted graph $G = (V, E, w)$, the graph transduction game (GTG) is as follow:

- Players = Objects;
- Pure strategies = Labels;
- Mixed strategies = Weighted labeling assignments;
- Payoffs = Compatibility coefficients

The transduction game is in fact played among the unlabeled players to choose their memberships. In this sense, the concept of **Nash equilibrium** can be put into one-to-one correspondence with **consistent labeling** of nodes, and it can be solved using **standard relaxation labeling / replicator dynamics**. Moreover, the framework can cope with symmetric, negative and asymmetric similarities and category-level similarities.

7.6.1 Word sense disambiguation

WSD is the task of **identifying** the **intended meaning of a word based on the context in which it appears**. It has been studied since the beginning of NLP and also today is a central topic of this discipline, since it is used in applications like text understanding, machine translation, opinion mining, sentiment analysis and information extraction.
The WSD problem can be formulated in **game-theoretic terms modeling**:

- The players of the games as the words to be disambiguated;
- The strategies of the games as the senses of each word;
- The payoff matrices of each game as a sense similarity function;
- The interactions among the players as a weighted graph.

In this case the **Nash equilibrium** correspond to **consistent word-sense assignments**!

- **Word-level similarities**: proportional to strength of co-occurrence between words;
- **Sense-level similarities**: computed using WordNet / BabelNet ontologies.

Picture 7.6.1 shows an example of a sentence, while 7.6.1 shows the iterations of the solution of the WSD problem.

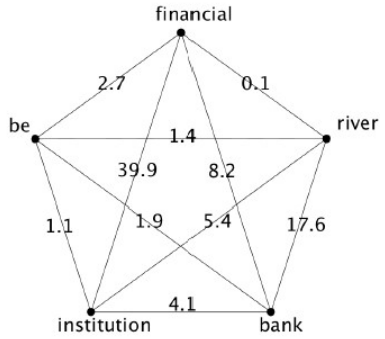


Figure 144: Example of graph for the graph-transduction problem

As we can see, initially, each of the senses of the words have the same probability, and then these probabilities are updated as the algorithm proceeds the execution.

7.6.2 The "protein function prediction" game

Network-based methods for the **automatic prediction of protein functions** can greatly benefit from exploiting both the similarity between proteins and the similarity between functional classes. **Hume's principle** states that **similar proteins** should have **similar functionalities**.
We envisage a (non-cooperative) game where:

- Players = proteins;
- Strategies = functional classes;

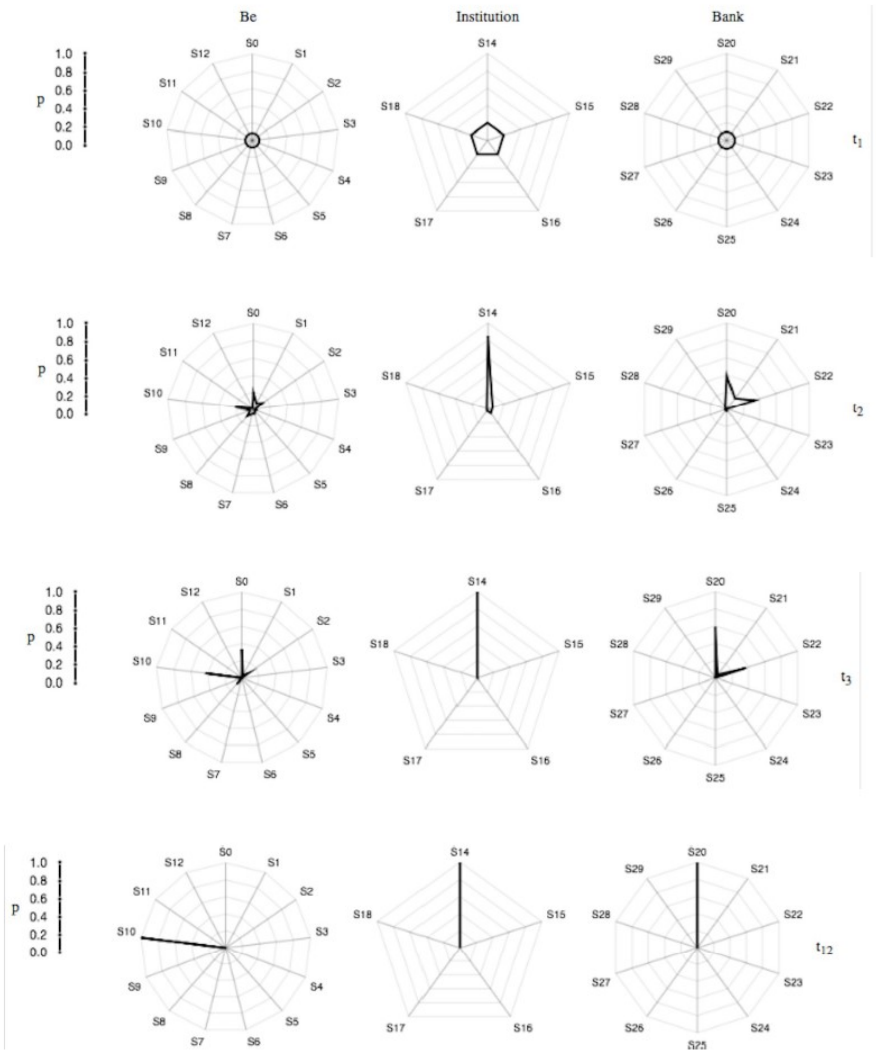


Figure 145: Example of WSD game dynamics

- Payoff function = combination of protein- and function-level similarities.

The **Nash equilibrium** turns out to provide **consistent functional labelings of proteins**.

7.7 Metric learning: triplet loss

We now present a system, called **FaceNet**, that directly learns a **mapping** from **face images** to a **compact Euclidean space** where **distances** directly **correspond** to a **measure of face similarity**. Once this space has been produced, tasks such as face recognition, verification and clustering can be easily implemented using standard techniques with FaceNet embeddings as feature vectors.
The problem of embedding an image into an Euclidean space is represented in Picture 7.7.

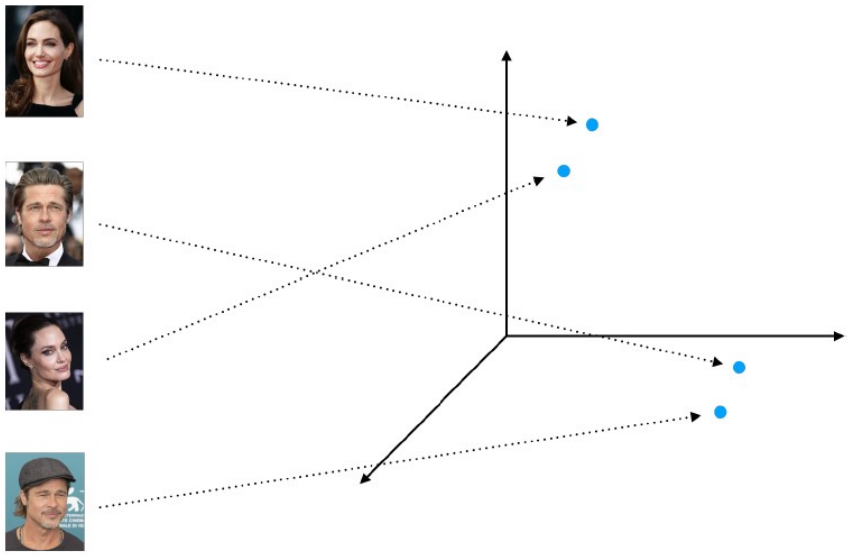


Figure 146: Embedding an image into an Euclidean space

7.7.1 The triplet loss

The concept on which the embedding is based is the so-called **triplet loss**. The idea is to take **3 instances** from the training set: an **anchor**, a **negative instance** (with different label w.r.t. the anchor) and a **positive instance** (with the same label of the anchor). Thus, the goal of the embedding is to:

- **Minimize** the **distance** between the **anchor** and the **positive** example;
- **Maximize** the **distance** between the **anchor** and the **negative** example.

A visual representation of the relationships between anchor, negative and positive examples is provided in Picture 7.7.1.
Picture 7.7.1 shows the **goal** of the embedding: the **distance** in the Euclidean space between the embedding representation of the anchor, negative and positive examples must be **minimized**.
In particular, the Picture shows the exact **loss** to be minimized. As we can see the loss is defined as

$$L_{\text{triplet}} = \max(d_p - d_n + \alpha, 0)$$

, which means that:

- If $d_p - d_n + \alpha \leq 0$, then the loss is equal to 0, as we desire;
- Otherwise, the loss has value $d_p - d_n + \alpha > 0$.

7.7.2 Pipeline

Picture 7.7.2 shows the **pipeline** of the triplet loss computation.
As we can see:

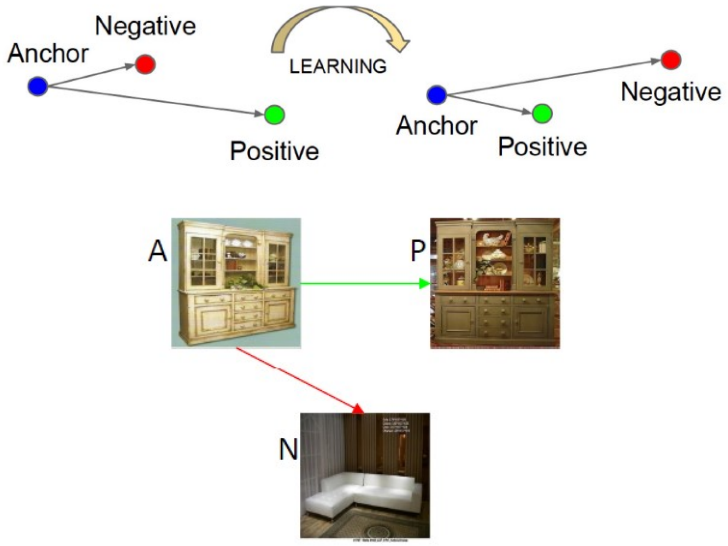


Figure 147: Anchor, negative and positive examples

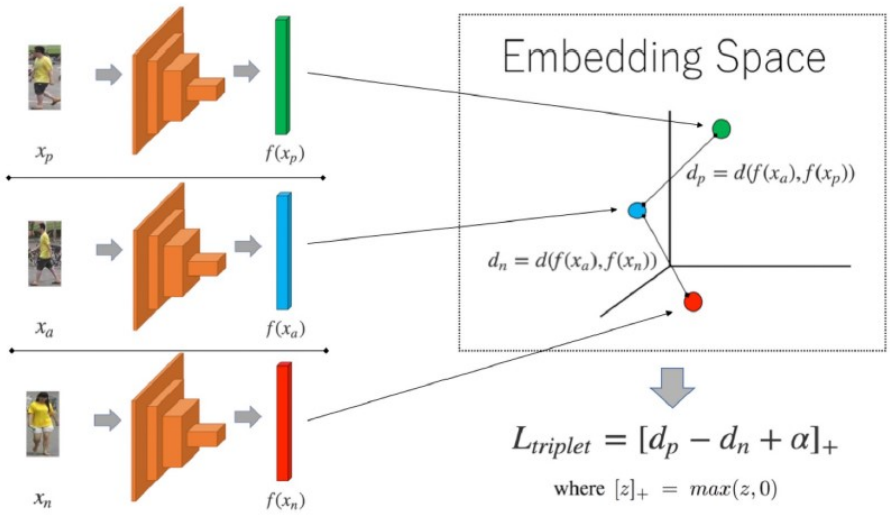


Figure 148: Triplet loss

- In the **first stage**, a **mini-batch** is sampled from the training data, which usually contains k identities with several images per identity;
- **Deep Neural Networks** are used to **learn** a **feature embedding**, e.g. a 128-dimensions feature vector;
- In the **third stage**, a subset of **triplets** are selected using some **triplet selection methods**;
- Lastly, the **loss** is evaluated using the selected triplets.

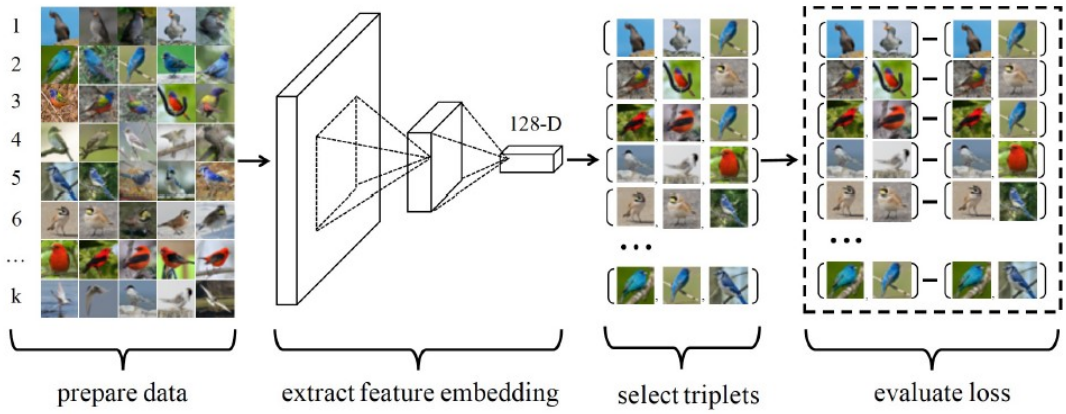


Figure 149: The triplet loss pipeline

7.7.3 The group loss

Clearly, we can **generalize** the concept of **triplet loss** by considering **all the relations among the nodes** contained in the mini-batch, and not only considering triplets. An example of **group loss** is provided in Picture 7.7.3.

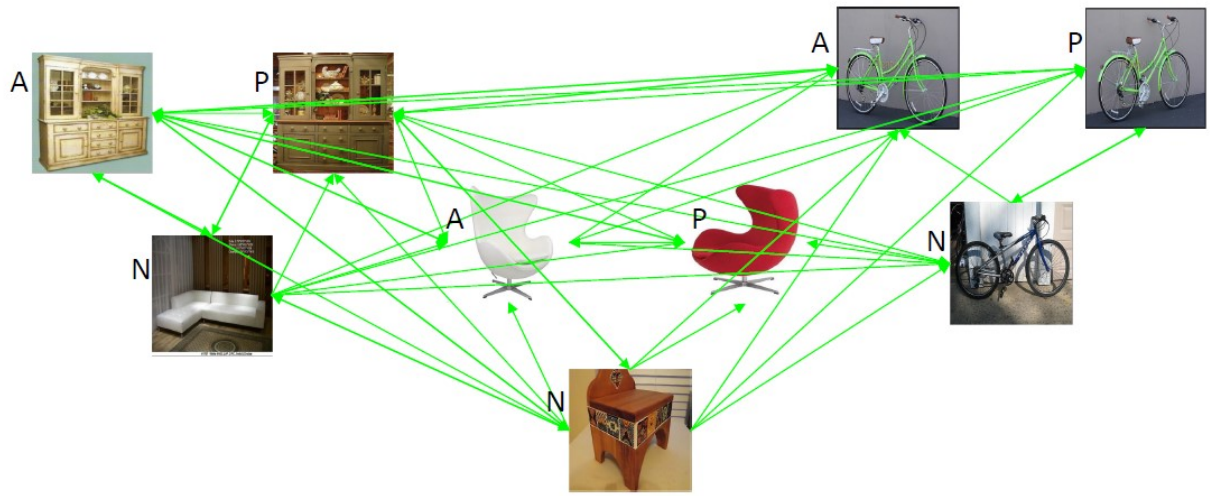


Figure 150: Group loss

Picture 7.7.3 shows the **pipeline** of this new approach.
As we can see:

- Again, **CNN's** are used to retrieve the **feature embeddings**. Notice that the different CNN's are characterized by the fact that they **share** the same set of **weights**;
- Then, we have three steps:

    1. **Initialization**: initialize X, the image-label assignment using the softmax outputs of the NN. Compute the $n \times x$ pairwise similarity matrix W using the NN embeddings;

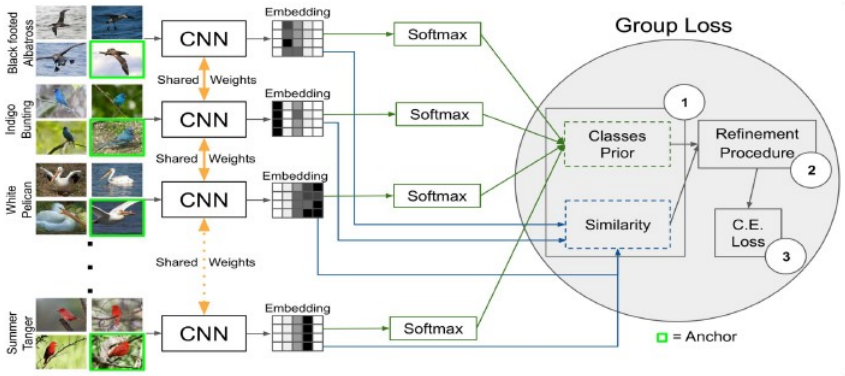


Figure 151: The group loss pipeline

2. **Refinement**: iteratively, refine X considering the similarities between all the mini-batches images, as encoded in W, as well as their labeling preferences;
3. **Loss computation**: compute the cross-entropy loss of the refined probabilities and update the weights of the NN using backpropagation.

As we can see, the **goal** of the **loss function** is to **refine** the **soft-labels** predicted by a neural network, using an **iterative procedure** based on the similarity between the images in the mini-batch.
Picture 7.7.3 shows a toy **example** of the refinement procedure, where the goal is to classify sample C based on the similarity with samples A and B.

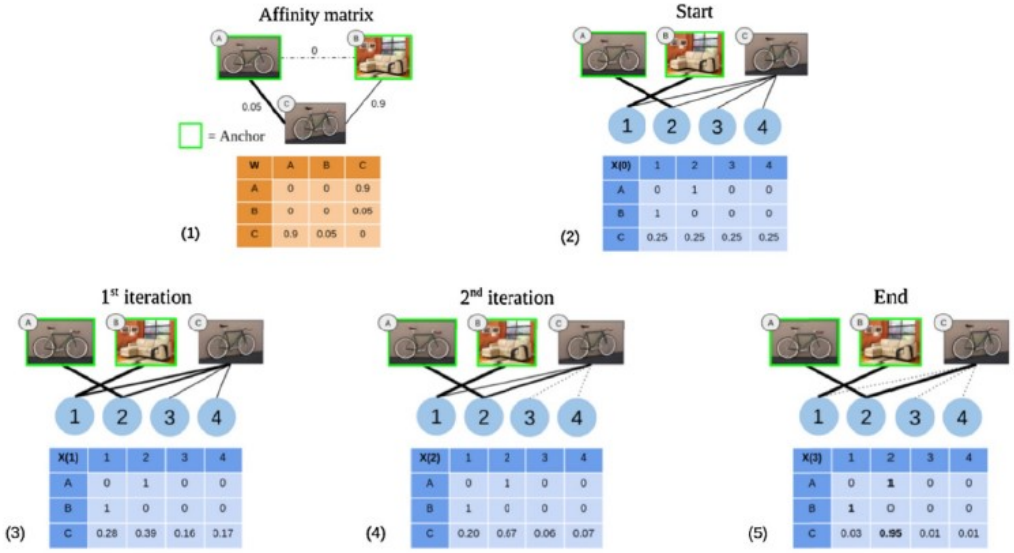


Figure 152: Results

(1) The affinity matrix used to update the soft assignments. (2) The initial labeling of the matrix. (3-4) The process iteratively refines the soft assignment of the unlabeled sample C. (5) At the end of the process, sample C gets the same label of A, (A, C) being more similar than (B, C).

References

[Dalal and Triggs(2005)] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 886–893. Ieee, 2005.

[Kotropoulos et al.(1994)Kotropoulos, Magnisalis, Pitas, and Strintzis] Constantine Kotropoulos, X Magnisalis, Ioannis Pitas, and Michael G Strintzis. Nonlinear ultrasonic image processing based on signal-adaptive filters and self-organizing neural networks. *IEEE Transactions on image processing*, 3(1):65–77, 1994.

[Leung et al.(1995)Leung, Burl, and Perona] Thomas K Leung, Michael C Burl, and Pietro Perona. Finding faces in cluttered scenes using random labeled graph matching. In *Proceedings of IEEE International Conference on Computer Vision*, pages 637–644. IEEE, 1995.

[Pavan and Pelillo(2006)] Massimiliano Pavan and Marcello Pelillo. Dominant sets and pairwise clustering. *IEEE transactions on pattern analysis and machine intelligence*, 29(1):167–172, 2006.

[Rowley et al.(1998a)Rowley, Baluja, and Kanade] Henry A Rowley, Shumeet Baluja, and Takeo Kanade. Neural network-based face detection. *IEEE Transactions on pattern analysis and machine intelligence*, 20(1):23–38, 1998a.

[Rowley et al.(1998b)Rowley, Baluja, and Kanade] Henry A Rowley, Shumeet Baluja, and Takeo Kanade. Rotation invariant neural network-based face detection. In *Proceedings. 1998 IEEE computer society conference on computer vision and pattern recognition (Cat. No. 98CB36231)*, pages 38–44. IEEE, 1998b.

[Sung and Poggio(1998)] K-K Sung and Tomaso Poggio. Example-based learning for view-based human face detection. *IEEE Transactions on pattern analysis and machine intelligence*, 20(1):39–51, 1998.

[Tesfaye et al.(2017)Tesfaye, Zemene, Prati, Pelillo, and Shah] Yonatan Tariku Tesfaye, Eyasu Zemene, Andrea Prati, Marcello Pelillo, and Mubarak Shah. Multi-target tracking in multiple non-overlapping cameras using constrained dominant sets. *arXiv preprint arXiv:1706.06196*, 2017.

[Vascon et al.(2016)Vascon, Mequanint, Cristani, Hung, Pelillo, and Murino] Sebastiano Vascon, Eyasu Z Mequanint, Marco Cristani, Hayley Hung, Marcello Pelillo, and Vittorio Murino. Detecting conversational groups in images and sequences: A robust game-theoretic approach. *Computer Vision and Image Understanding*, 143:11–24, 2016.

[Viola and Jones(2001)] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, volume 1, pages I–I. Ieee, 2001.

[Yang and Huang(1994)] Guangzheng Yang and Thomas S Huang. Human face detection in a complex background. *Pattern Recognition*, 27(1):53–63, 1994. ISSN 0031-3203. doi: https://doi.org/10.1016/0031-3203(94)90017-5. URL `https://www.sciencedirect.com/science/article/pii/0031320394900175`.

[Zemene et al.(2018a)Zemene, Tesfaye, Idrees, Prati, Pelillo, and Shah] Eyasu Zemene, Yonatan Tariku Tesfaye, Haroon Idrees, Andrea Prati, Marcello Pelillo, and Mubarak Shah. Large-scale image geolocalization using dominant sets. *IEEE transactions on pattern analysis and machine intelligence*, 41(1):148–161, 2018a.

[Zemene et al.(2018b)Zemene, Alemu, and Pelillo] Eyasu Zemene Zemene, Leulseged Tesfaye Alemu, and Marcello Pelillo. Dominant sets for "constrained" image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 41(10):2438–2451, 2018b.