POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Aerospaziale

Tesi di Laurea Magistrale

Design and implementation of a pilot non-dependent tracking system

Relatore
Prof. Paolo Maggiore

Candidato
Alice Serra

Ottobre 2019

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The work proposed in this thesis is the outcome of a collaboration with Digisky, an Italian company that works on aerial ground monitoring technologies for Earth preservation, emergency management and surveillance scopes.
The company flagship product is called *SmartBay*. *SmartBay* is a cutting edge interface that allows for "plug-and-play" of various sensors under the aircraft wing.
The platform has three payload slots (figure 1.1), allowing for different recording tools and sensors to operate at the same time and weighting up to 40 kilograms overall. Examples of high performance payloads that can be embarked on the aircraft are video cameras with gyro-stabilization, chemical sensors for air quality check and custom sensors as scatterometer systems.

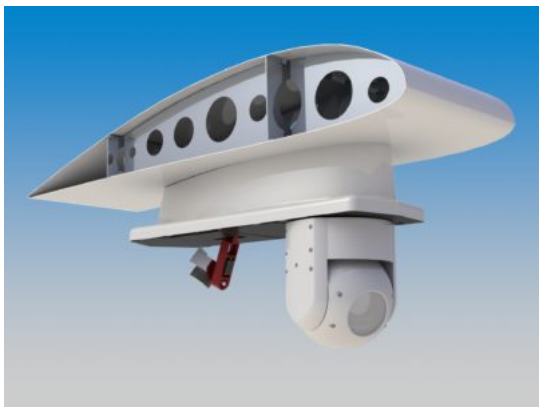

Figure 1.1: SmartBay platform and the three payloads slots

The aircraft equipped with *SmartBay* is ready to perform different flight operations according to the "smart" configuration of the mission. For instance, using the proven Tecnam P92 as the base platform, the *Tecnam P92 SmartBay* aircraft accommodates a wide range of avionics while keeping smooth aerodynamics lines and light weight.

Figure 1.2: The Tecnam P92 Eaglet SmartBay

The whole SmartBay system aims at being easily controllable so that high results in term of accuracy and full safety standards are still guaranteed. To accomplish this goal, the on-board SmartBay systems includes a touchscreen tablet that allows the pilot to drive payload sensors while keeping high safety awareness.

Figure 1.3: The on-board SmartBay

1.1 Thesis Objective

As mentioned above, the platform can carry any type of sensors. For this reason, the company aims at producing a proprietary sensor: the *Smart Gimbal*. The sensor is designed to be a 10-inches gimbal camera holder, damped, stabilized, tilt and yaw swiveling that can support professional video cameras, weighting up to 3 kg.

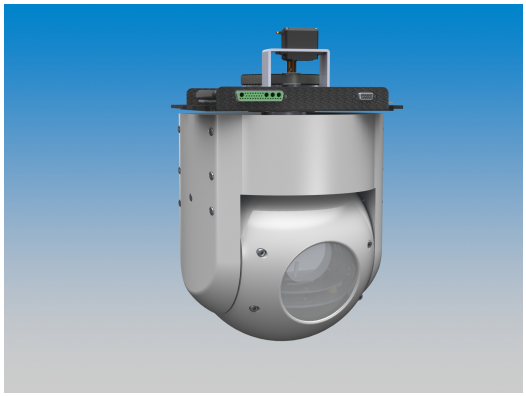

Figure 1.4: The *Smart Gimbal* sensor

Among the purposes the *Smart Gimbal* is designed for is that of performing objects tracking for search and rescue or surveillance scopes. With respect to guarantee easy control and safety awareness, the work presented hereafter is focused on the design and implementation of a pilot non-dependent tracking system that aims at performing the autonomous detection and tracking of a specific target during a single pilot flight mission. To start, the work is designed to detect and track one single target at a time (a car or a truck) with the future purpose of extending the field of application to several objects classes and numbers.

Chapter 2

Object tracking

Tracking is the problem of making a prediction about the motion of an object given a sequence of images. Many are the applications of this task:

- **recognition from motion**. The motion of objects is quite peculiar. It may be possible to identify an object from its motion and be able to tell what it is doing.
- **surveillance**. Understand what an object is doing can be very useful. Let's consider this example. Different kinds of trucks should move in different, fixed patterns in an airport and there are combinations of motions that should never occur (no truck should ever stop on an busy runway). For these reasons, it would be useful to have a computer system that can monitor the activities and send a warning if it detects a dangerous case.
- **targeting**. A consistent part of tracking applications is oriented towards deciding what to shoot and hitting it. Typically the literature describes tracking using radar or infra-red signals rather than vision, but the principal issues are the same.

2.1 Related Work

Many approaches have been proposed in the literature for object tracking [1]. Generally, in typical tracking problems some set of measurements from a sequence of images are available along with a model for the object's motion. These measurements could be, for instance, the position of some image points at a certain moment. Not all are guaranteed to be relevant, in the sense that some could come from the object of interest and some might come from other objects, or from noise.
In the next sections, the main algorithmics of object tracking will be exposed. We will start with considering tracking as a probabilistic problem.

2.1.1 Tracking as a Probabilistic Inference Problem

A moving object can be modeled as having some internal state that includes information about its motion; the state of the object at the i'th frame is referred to as X_i. The measurements obtained in the i'th frame are grouped in a random variable Y_i where y_i will stand for the value of a measurement. The problem can be split in three sub-problems which are:

- **Prediction**: what does the set of previous measurement y_0, ... , y_{i-1} predict for the i'th frame? To answer this question, a representation of the probability density P of the state X_i is obtained – $P(X_i \mid Y_0 = y_0, \ldots, Y_{i-1} = y_{i-1})$.
- **Data association**: some of the measurements obtained from the i'th frame may give information about the object's state. Again, to identify these measurements we use $P(X_i \mid Y_0 = y_0, \ldots, Y_{i-1} = y_{i-1})$.

- **Correction**: once we have relevant measurements Y_i we can compute the representation $P(X_i \mid Y_0 = y_0, \ldots, Y_i = y_i)$ can be computed.

While performing object tracking, the following assumptions are introduced in order to simplify the discussion:

- **Only the immediate past matters**: formally this means that

$$P(X_i \mid X_1, \ldots, X_{i-1}) = P(X_i \mid X_{i-1})$$

  As it will be illustrated, this assumption simplifies the structure of the algorithms.

- **Measurements depend only of the current state**: Y_i is assumed to be conditionally independent of all other measurements given X_i.

As we can infer, the key algorithmic issue involves finding the representation of a relevant probability density of the state X_i . The simplest scenario occurs when the dynamics are linear, the measurement model is linear and the noise model are Gaussian.

2.1.2 Linear Dynamic Models and Kalman Filter

Under linear assumptions, the state is considered to advance by multiplying it by some known matrix (which my depend on the frame) and then adding a normal random variable of zero mean μ and know covariance Σ. Formally, notation is

$$x \sim N(\mu, \Sigma)$$

which means that x is the value of a random variable with a normal probability distribution with mean μ and covariance Σ. Given so, the dynamic model can be written as

$$x_i \sim N(D_i x_i, \Sigma_{d_i})$$

$$y_i \sim N(M_i x_i, \Sigma_{m_i})$$

where x_i and y_i are the state and the measurement at the i'th frame respectively. The indexes highlight that the covariances and the matrices could vary from frame to frame.

Constant Velocity

Let assume that the vector p gives the position and v the velocity of a point moving with constant velocity. In this scenario, $p_i = p_{i-1} + \Delta t v_{i-1}$ and $v_i = v_{i-1}$. This means that the position and the velocity of the point can be stack into a single state vector

$$x = \begin{Bmatrix} p \\ v \end{Bmatrix}$$

and

$$D_i = \begin{Bmatrix} Id & (\Delta t)Id \\ 0 & Id \end{Bmatrix}$$

where $D_i = Id$ would mean that the point is moving under random motion – its new position is its old position, plus some Gaussian noise term. According to this formulation, it may appear that the object tracked has a stationary motion. However, it is commonly used for objects for which no better dynamic model is known – the random component is assumed to be quite large.
In many cases it is expected that

$$M_i = \begin{Bmatrix} Id & 0 \end{Bmatrix}$$

meaning that the model is only looking at the position of the point since it is assumed that it is moving with constant velocity.
In figure 2.1 a constant velocity dynamic model for a point on a line is shown. In this scenario, the state space is two dimensional – one coordinate for position, one for velocity. The figure on the **top left** shows a plot of the state, each asterisk stands for a different state. The vertical axis which is the velocity shows some small variation compared with the horizontal axis (position). This small change is due to the random component of the model, so that the velocity is constant up to a random change. The figure on the **top right** shows the first component of the state (position) plotted against the time axis. It can be noticed that the point is moving with approximately constant velocity. The figure of **bottom** overlaps the measurements which are the circles on this last plot. Even if they appear quite poor, this does not significantly hamper the ability to track.

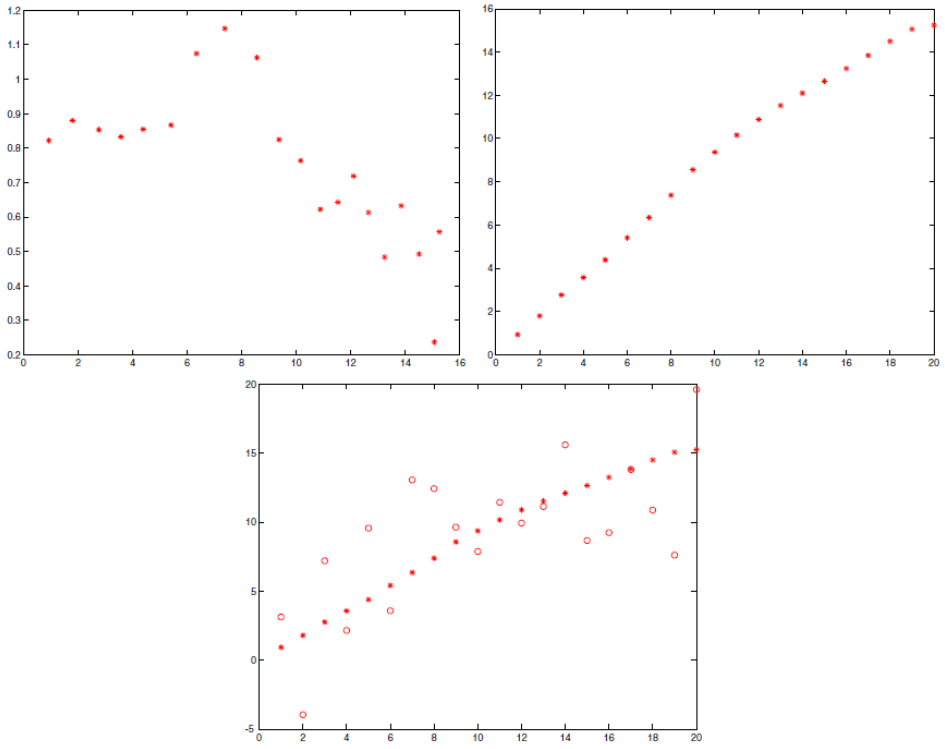

Figure 2.1: A constant velocity dynamic model

Constant Acceleration

Let know assume that the vector p gives the position, vector v the velocity and vector a the acceleration of a point moving with constant acceleration. In this case, $p_i = p_{i-1} + \Delta t v_{i-1}$, $v_i = v_{i-1} + \Delta t a_{i-1}$ where $a_i = a_{i-1}$. Again, the position, velocity and acceleration can be stack into a single state vector

$$x = \begin{Bmatrix} p \\ v \\ a \end{Bmatrix}$$

and

$$D_i = \begin{Bmatrix} Id & (\Delta t)Id & 0 \\ 0 & Id & (\Delta t)Id \\ 0 & 0 & Id \end{Bmatrix}$$

Again, in many cases it is expected that

$$M_i = \begin{Bmatrix} Id & 0 & 0 \end{Bmatrix}$$

meaning that the model is only looking at the position of the point since it is assumed that it is moving with constant acceleration.

In figure 2.2 a constant acceleration dynamic model for a point on a line is shown. On the left, the first two components of state – the position on the x-axis and the velocity on the y-axis – are plotted and they are expected to look like (t^2, t), and they does. On the right, a plot of the position against time is shown and it can be noticed that the point is moving away from its start position increasingly quickly.

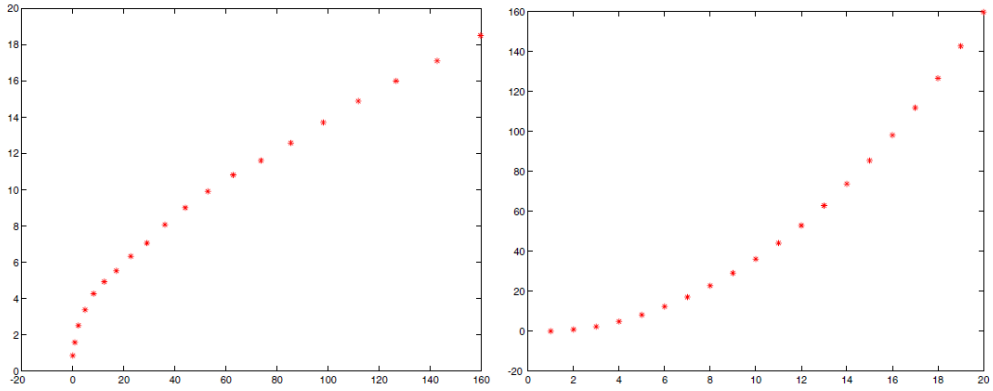

Figure 2.2: A constant acceleration dynamic model

Kalman Filtering for a 1D State Vector

An important feature of the class of models illustrated is that all the conditional probability models are normal. In particular, $P(X_i \mid y_1, \ldots, y_{i-1})$ is normal, and so is $P(X_i \mid y_1, \ldots, y_i)$. This feature makes them easy to be represented. In fact, the model will admit a relatively simple process where all we need to do is maintain representations of the mean and the covariance for the prediction and correction phase. The dynamic model is now

$$x_i \sim N(d_i x_{i-1}, \sigma^2_{d_i})$$

$$y_i \sim N(m_i x_i, \sigma^2_{m_i})$$

where σ is the standard deviation.
For the representation of $P(X_i \mid y_0, \ldots, y_{i-1})$ and $P(X_i \mid y_0, \ldots, y_i)$, the following annotation is adopted:

- the **mean** of $P(X_i \mid y_0, \ldots, y_{i-1})$ and of $P(X_i \mid y_0, \ldots, y_i)$ are represented as $\overline{X_i}^-$ and $\overline{X_i}^+$ where the subscripts stands for the estimation of X_i immediately before and after the i'th measurements arrives.
- similarly, the **standard deviation** of $P(X_i \mid y_0, \ldots, y_{i-1})$ and of $P(X_i \mid y_0, \ldots, y_i)$ are σ_i^- and σ_i^+

It will be assumed that $P(X_{i-1} \mid y_0, \ldots, y_{i-1})$ are known so that $\overline{X}^+_{i-1}$ and σ^+_{i-1}.
Given this assumption, it can be demonstrated that the 1D Kalman filter updates estimates of the mean and covariance of the various distributions encountered while tracking a one-dimensional state variable using the dynamic model shown in figure 2.3.

Dynamic Model:

$$x_i \sim N(d_i x_{i-1}, \sigma_{d_i})$$

$$y_i \sim N(m_i x_i, \sigma_{m_i})$$

Start Assumptions: $\overline{x}_0^-$ and σ_0^- are known

Update Equations: Prediction

$$\overline{x}_i^- = d_i \overline{x}_{i-1}^+$$

$$\sigma_i^- = \sqrt{\sigma_{d_i}^2 + (d_i \sigma_{i-1}^+)^2}$$

Update Equations: Correction

$$x_i^+ = \left(\frac{\overline{x}_i^- \sigma_{m_i}^2 + m_i y_i (\sigma_i^-)^2}{\sigma_{m_i}^2 + m_i^2 (\sigma_i^-)^2} \right)$$

$$\sigma_i^+ = \sqrt{\left(\frac{\sigma_{m_i}^2 (\sigma_i^-)^2}{(\sigma_{m_i}^2 + m_i^2 (\sigma_i^-)^2)} \right)}$$

Figure 2.3: The 1D Kalman filter updated estimates

2.1.3 Kalman filter limitation and drawbacks

To sum up, the core idea of Kalman filter is to use the available measurements and previous predictions to arrive at a best guess of the current state, while keeping the possibility of errors in the process. So, it essentially infers a new distribution (the predictions) from the previous state distribution and the measurement distribution as figure 2.4 shows.

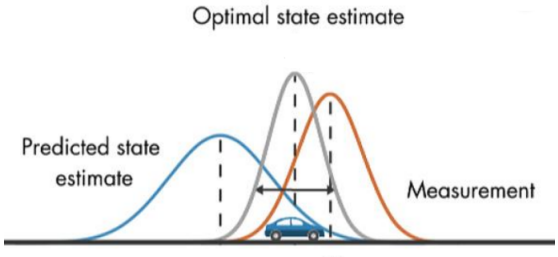

Figure 2.4: The Kalman Filter process

As it has been illustrated in the previous section, Kalman filter works best for linear systems with Gaussian processes involved. However, in our case the tracks can leave the linear realm and the problem may not be suited for the use of Kalman filters. It is obvious that we cannot expect that the object we want to track (for instance a car or a truck) would follow a constant velocity model (the motion may be uncertain, with abrupt acceleration, slowing downs or stops). In this scenario, the noise associated with the constant velocity model that we assume cars would follow increases exponentially. Moreover, we need to consider that in our project the camera used for tracking is also in motion with respect to the object. This aspect can often lead to unintended consequences. Since many trackers as Kalman filter consider the features

from an object to track them, such tracker might fail in scenarios where the object appears different because of the camera motion (appear bigger or smaller).
Furthermore, even if we should be able to optimize these complex algorithms the bottleneck of our system would always be the slow movement of the camera which will not be able to keep up with the tracking algorithm.
Given so, to the best of my knowledge a more robust and customized tracking system have to be implemented for the purpose of this project.

2.2 Proposed tracking system

To overcome the difficulties mentioned in previous section, a single object tracking system via detection and bounding box progression is proposed. In other words, object detection will be performed as first step, followed by an across-frame track algorithm.
Before giving a deeper insight about object detection and how it is performed, I will illustrated hereafter the across-frame track algorithm that will be implemented in this project.

2.2.1 Across-frame track algorithm overview

Let's assume we apply an object detection algorithm to a single frame of our entire video. We will see in chapter 3 that, if the object of interest is within the frame, object detection returns a bounding box around the target along with the following information:

- x_t which is the coordinate in pixels in the x-direction of the **top left** corner of the bounding box;
- y_t which is the coordinate in pixels in the y-direction of the **top left** corner of the bounding box;
- h which is the **height** of the bounding box;
- w which is the **width** of the bounding box.

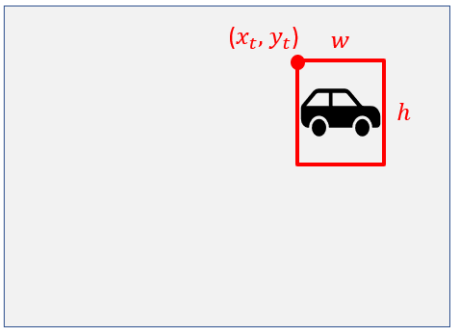

Figure 2.5: Object detection output – the bounding box and its coordinates

Ideally, to track the target we would like the camera to point always at the center of the bounding box. If the center of the image and that of the bounding box do not coincide, the camera has to rotate on its pitch and yaw axis (figure 2.6) in order to compensate the mismatch. The result of the rotation is shown in figure 2.7 where the two centers are led to coincide.

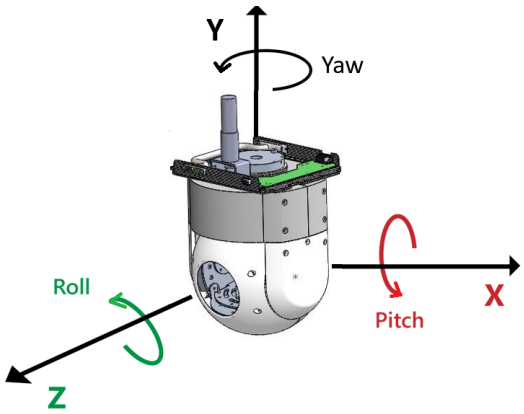

Figure 2.6: Gimbal rotation on its pitch and yaw axis

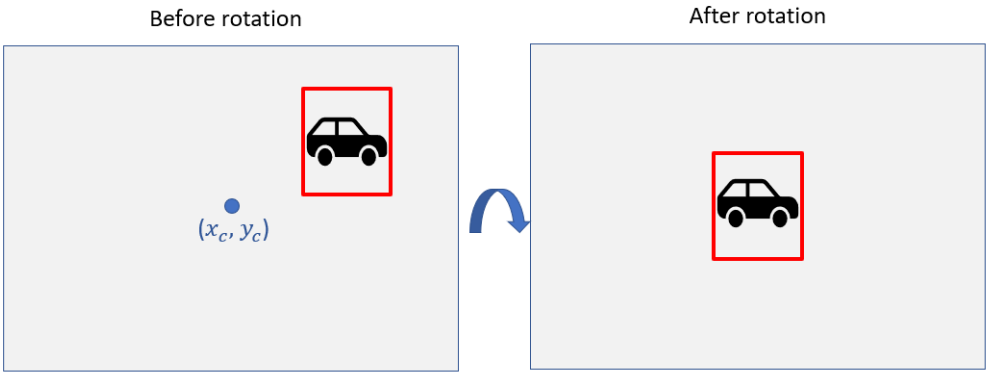

Figure 2.7: Mismatch compensation to track the target

To to build the across-frame track algorithm, a brief insight into some camera geometric parameters is presented in the next section.

Geometric Camera Parameters

In photography, among the primary descriptor of lenses is **focal length**. Focal length, usually represented in millimeters (mm), is an optical distance from the point where light rays converge to form a sharp image of an object to the digital sensor in the camera.

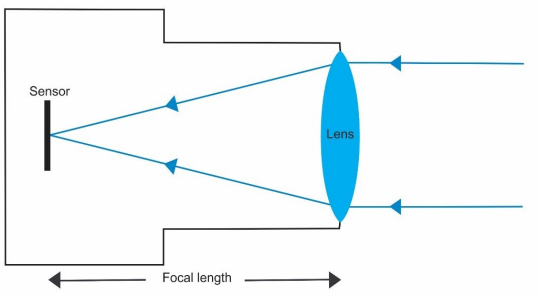

Figure 2.8: Focal length

From focal length it is possible to derive the **angle of view** which is the angular extent of a given scene that is imaged by a camera – in other words it describes how much of the scene will be captured. Focal length f and angle of view θ of a lens are

inversely proportional according to the following formula

$$\theta = 2 \arctan \frac{d}{2f} \tag{2.1}$$

where d is the dimension of the camera sensor.
Let's consider the following diagram which illustrates the relationship between the focal length and the angle of view. In line with the previous equation, it can be see that the longer the focal length, the narrower the angle of view and vice versa.

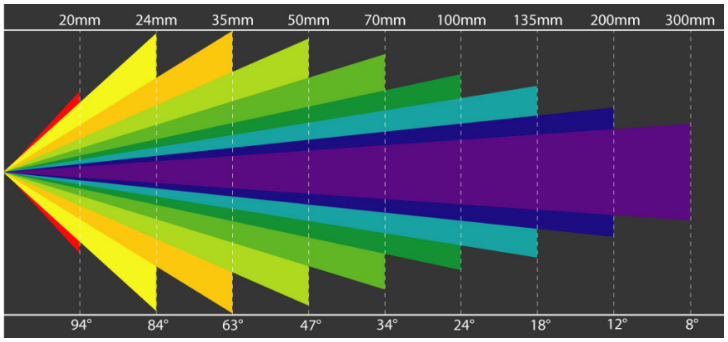

Figure 2.9: Focal length and angle of view

The angle of view of a camera can be measured *horizontally, vertically, or diagonally.* As it can be noticed from equation (2.1), this angle depends not only on the lens, but also on the dimension d of the sensor used. The two major types of digital image sensor are charge-coupled devices (CCD) and complementary metal–oxide–semiconductor (CMOS). Regardless the technology, an image sensor is usually rectangular (we indicate with the base with b and the height with$v$) and can have different size. Depending on the dimension that we choose, the angles of view can be measured:

- horizontally, $\theta_h = 2 \arctan \frac{b}{2f}$
- vertically, $\theta_v = 2 \arctan \frac{v}{2f}$
- diagonally, $\theta_d = 2 \arctan \frac{d}{2f}$

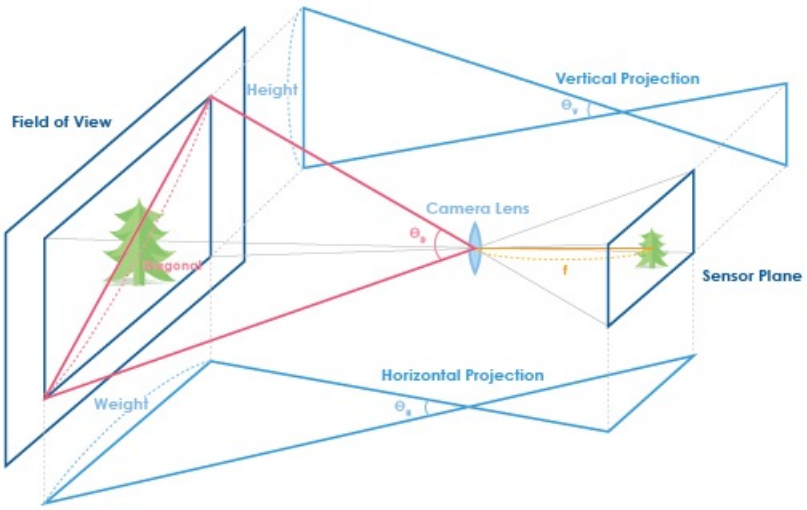

Figure 2.10: Horizontal, vertical and diagonal field of view

2.2.2 Tracking Algorithm Implementation

In section 2.2.1 we saw that object detection returns a bounding box that can be identified through 4 coordinates (x_t, y_t, h and w). By knowing this information, we can immediately get the coordinates of the center of the bounding box which are:

$$x_{b_c} = x_t + \frac{h}{2} \tag{2.2}$$

$$y_{b_c} = y_t + \frac{w}{2} \tag{2.3}$$

After that, it is possible to compute the difference between the center of the bounding box (x_{b_c},y_{b_c}) and the center of the image frame (x_{f_c},y_{f_c}) along the x and the y direction by simple subtraction:

$$dx = x_{b_c} - x_{f_c} \tag{2.4}$$

$$dy = y_{b_c} - y_{f_c} \tag{2.5}$$

Ideally, the frame can be split in four quarters: we can assume that the numeration is anticlockwise and the first quarter corresponds to the top right region of the frame as shown in the sketch below.

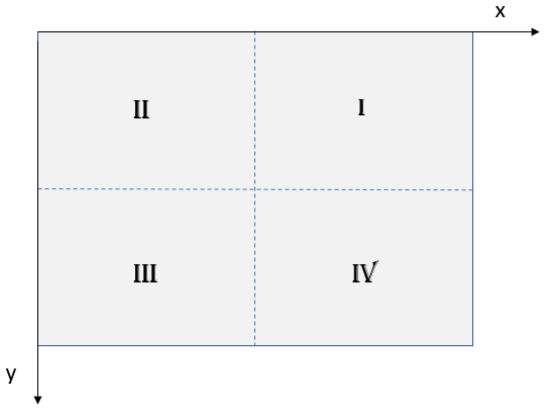

Figure 2.11: The image frame quarters

According to equations 2.4 and 2.5, there are four possible scenarios:

- $dx > 0$ and $dy > 0$: the bounding box (and thus the target) is in the **first** quarter;
- $dx < 0$ and $dy > 0$: the bounding box is in the **second** quarter;
- $dx < 0$ and $dy < 0$: the bounding box is in the **third** quarter;
- $dx > 0$ and $dy < 0$: the bounding box is in the **fourth** quarter;

As stated before, in order to track the target we want the camera to rotate around the pitch and yaw angle so that the center of bounding box and that of the frame are overlapped. Given so, the algorithm has to return two angles that have to be provided to the 2-axis control system. The two angle are θ_{h_m} and θ_{v_m} where the subscripts h_m and v_m stand for horizontal and vertical mismatch. Precisely, θ_{h_m} represent the rotation on the **yaw axis** while θ_{v_m} the one on the **pitch axis**. To compute these two angles, the algorithm takes into consideration the triangles shown in figure 2.12 and sets the following proportions

$$dx : \tan\theta_{h_m} = \frac{h_w}{2} : \tan\frac{\theta_h}{2} \tag{2.6}$$

$$dy : \tan\theta_{v_m} = \frac{v_h}{2} : \tan\frac{\theta_v}{2} \tag{2.7}$$

where

- θ_h and θ_v are the horizontal and vertical field of view of the camera that we can easily get according to (2.1) once we know the camera specifications;
- h_m and v_h are the horizontal width and the vertical height **of the frame** and are measured in pixel;
- θ_{h_m} and θ_{v_m} are the unknown;

We can reformulate (2.6) and (2.7) as

$$\tan\theta_{h_m} = \frac{dx \cdot \tan\frac{\theta_h}{2}}{\frac{h_w}{2}}$$

$$\tan\theta_{v_h} = \frac{dy \cdot \tan\frac{\theta_v}{2}}{\frac{v_h}{2}}$$

and get θ_{h_m} and θ_{v_m} which are the two angles that have to be provided to the 2-axis control system in order to track the target. Depending on the sign of dx and dy, the angles can be either positive or negative and thus the rotation on the pitch and yaw axis.
Details about how the control system handles the tracking algorithm output will be given in chapter 5.

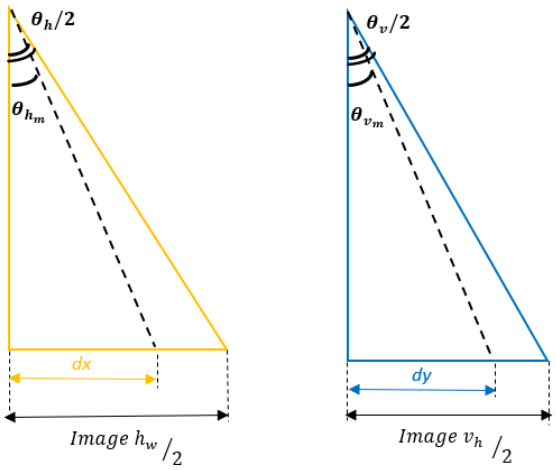

Figure 2.12: Tracking algorithm output θ_{h_m} and θ_{v_h}

Ideally, we would like to update θ_{h_m} and θ_{v_h} for each frame of the entire video but as we will see in chapter 5 this is not achievable due to the high computation cost required by object detection and because of the inertia of the handling system.

Chapter 3

Object Detection

In section 2.2 we mentioned that our system is designed to perform object detection across-frames in order to track the object. For this reason, this chapter includes a deep insight about object detection principles and approaches.

3.1 Object Detection Overview

Object detection involves detecting instances of objects from a particular class in an image. In other words, it can be see as the process of performing some *image classification* along with *object localization*.
Let's assume we want to build an airplane-helicopter classifier. **Image classification** takes an image as a input and predicts the class of the object in the image as shown in figure 3.1

Figure 3.1: Image classification

However, it will not predict the location of the airplane or helicopter within the image. The problem of identifying the location of an object (given its class) is called **localization**. Along with predicting the class of the object, we now have to predict the class as well as a rectangle (called bounding box) which contains the object itself. We saw that we need four information to uniquely identify a rectangle

Figure 3.2: Image classification and object localization

In conclusion for each object detected in the image, object classification should return the following information:

- object class name – every object class has its own special features that helps in classifying the class – for example all squares have equal side lengths and perpendicular corners. We will see that object detection uses these special features to look for squares in an image.
- bounding box top left x coordinate x_t;
- bounding box top left y coordinate y_t;
- bounding box height h;
- bounding box width w;

3.2 Object detection methods

Historically, object detection methods fall into **machine learning**-based approaches or **deep learning**-based approaches [2]. Machine Learning approaches first define features using one of the methods listed below and then use a technique such as **support vector machine** (SVM) to do the classification. On the other hand, deep learning techniques are able to do end-to-end object detection and are typically based on **convolutional neural networks** (CNN).

1. Machine Learning approaches
    - Viola–Jones object detection framework based on Haar features;
    - Scale-invariant feature transform (SIFT);
    - Histogram of oriented gradients (HOG) features;
2. Deep Learning approaches
    - Region Proposals (R-CNN, Fast R-CNN, Faster R-CNN);
    - Single Shot Multi Box Detector (SSD);
    - You Only Look Once (YOLO);

In order to illustrate the approaches listed above, a theoretical review about the broad concept of artificial intelligence, machine learning and deep learning principles it is provide in the following sections.

3.3 Artificial Intelligence

Inventors have long dreamed of designing machines able to think. When programmable computers were created, people wondered to know whether such devices might become intelligent. Nowadays, the ability gained by computer systems of simulating human intelligence is called **Artificial Intelligence** (AI).

In its early days, AI tackled and solved problems that might be difficult for human beings but relatively simple for computers. For instance, IBM's Deep Blue chess-playing computer easily beat then-reigning World Champion Garry Kasparov in 1997 [3]. In fact, even if planning a successful chess strategy might be a challenging goal, chess can be completely described by a brief list of mathematical rules.
Since its early days, the true challenge to artificial intelligence appeared to be tackling those tasks that people can easily perform but hardly describe such as recognizing faces in images. This scenario required computers to have much more knowledge of the world and suggested that AI systems needed the ability to acquire their own knowledge. This capability of a system to learn from itself is known as **machine learning**.

3.4 Principles of Machine Learning

Machine Learning can be defined as the science of getting computers to learn by themselves by extracting patterns from raw data. As the name suggests, machine learning learns about some training data that it takes as input and predicts a model based on these data

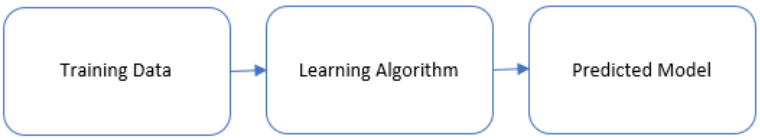

Figure 3.3: Training Process

Once the model has been trained, it can be applied to unseen data to make a prediction with the goal that it also generalizes to the new input data. This is the process through which we can assess the accuracy of the model.

Figure 3.4: Inference Process

3.4.1 Types of Machine Learning

Machine learning methods can be classified as *unsupervised and supervised depending* on the "knowledge" they are allowed to have during the training process. In **supervised learning** both input and desired output data are provided. In this case, the learning consists of training the model until it produces an output that corresponds to the desired output and the error is minimized. In image processing, for instance, an AI system must be provided with labeled pictures of some object classes so that, after a significant amount of observation, the system is able to correctly categorize unlabeled images of the same type.
Supervised learning is typically done in the context of *classification*, which consists of identifying to which category the input data belongs, or *regression*, which consists of predicting a continuous quantity output given some input values.

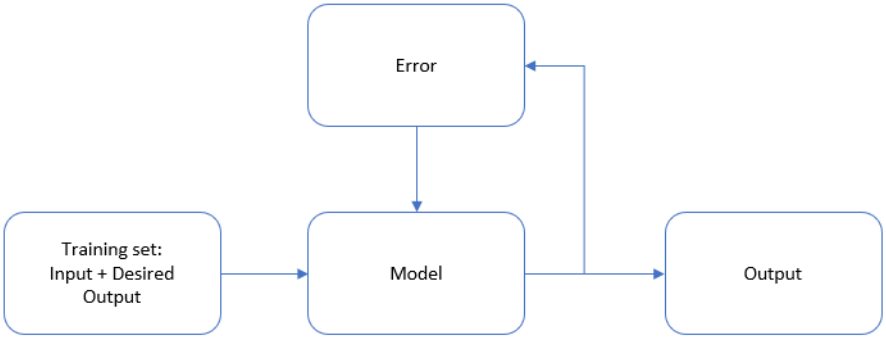

Figure 3.5: Supervised Learning

On the other hand, **unsupervised learning** algorithms experience an unlabeled dataset which may contain many features. One of the main applications used of unsupervised learning is cluster analysis. Cluster analysis consists of splitting unlabeled data into similarity groups called clusters.

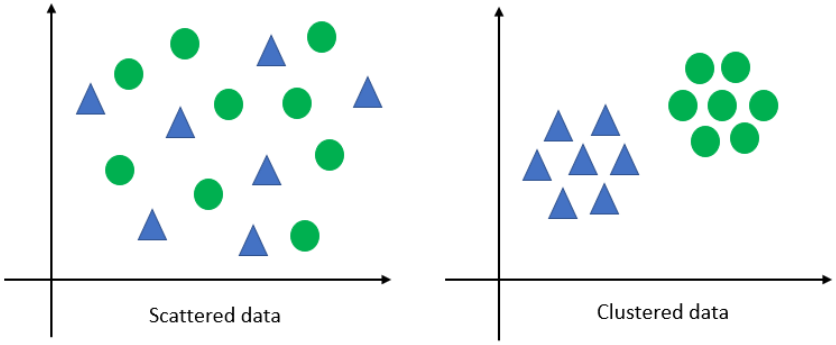

Figure 3.6: Cluster Analysis

3.5 Deep Learning

As we stated in the introduction to this chapter, the true challenge to AI is to solve problems that are intuitive but hard for people to describe formally. This solution is to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts. By gathering knowledge from experience, this approach avoids the need for human operators to formally specify all the knowledge that the computer needs. The hierarchy of concepts enables the computer to learn complicated concepts by building them out of simpler ones. If we draw a graph showing how these concepts are build on top of each other, the graph is deep, with many layers. For this reason the approach to machine learning is called **deep learning**.

3.5.1 Artificial Neural Networks

Deep learning tries to reproduce human brain and to mimic how it operates. Human brain is the most powerful tool on this planet for learning, adapting skills and applying them. If computers could copy that, then we could just leverage what natural selection has already decided for us.
In the human brain there are approximately 100 billion neurons and each neuron is connected to as many as about thousand of its neighbors. How is this recreate in a computer? This is done by creating an artificial structure call **artificial neural net** where there are many nodes also called **neurons**.

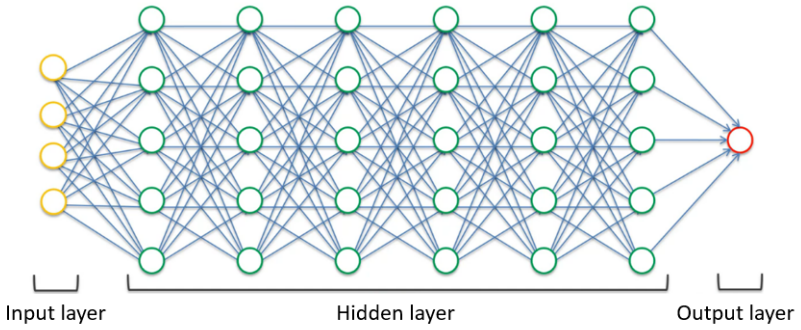

Figure 3.7: Input, Hidden and Output layers

The net has some neurons for input value that we know about a certain situation. In fact, when we want to predict something we always need to provide some input to start the prediction off. The whole of input neurons is called the **input layer**. In the analogy of human brain the input layer is our senses so whatever humans can see, hear or taste. Then , there is the **output layer** which is the what the net is supposed to predict. In between, there are the **hidden layers** where all the information from the input layer is coming through before reaching the output layer.

The Neuron

Now, let's move on to how are neurons created and represented in machines. Every neurons or node in the net has some input signals and some output signals that are basically values. The input values can come from either the input layer or from the hidden layer. Depending on the scenario, the input values can be *standardized independent variables or normalized independent variables* – that is because we want all variables to be quite similar in about the same range of values so that it will be easy for the neural networks to process them.

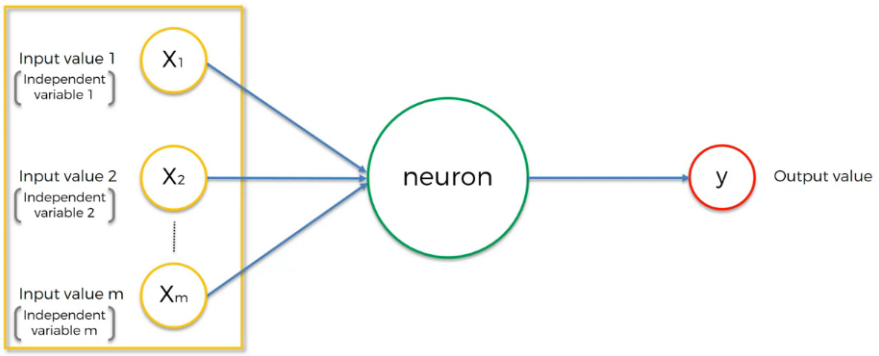

Figure 3.8: Neuron's connections

Concerning the output value, output value can be:

- **continuous**;
- **binary** – yes or no;
- **categorical** – in this case, the output value will not be just one – there will be several output values, one for each categories;

Another important concept is that of "synapse" which is the way the values are traveling through the net. All synapses get assigned weights w which play an important role in the learning process. From weights, the net can establish what signal is poor and what is not important to a certain neuron, what signal gets passed along and so on. So weights are crucial and they are the elements that get adjusted under the artificial neural networks learning process.

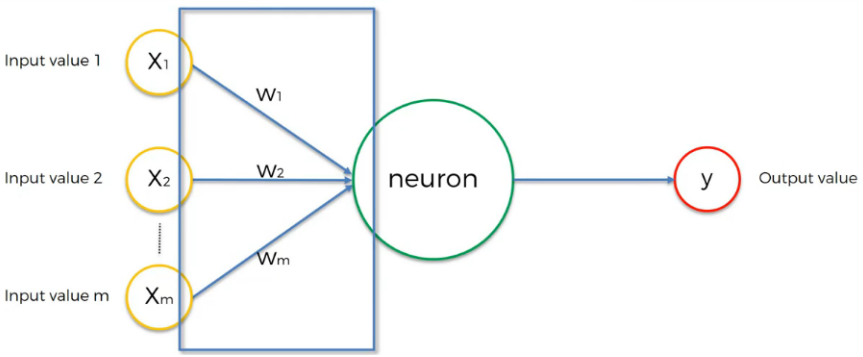

Figure 3.9: Artificial Neural Network weights

Once the signals with their weights reach the neuron, they get added up. After that, the neuron applies an *activation function* ϕ to the weighted sum – it is basically a function assigned to the neuron which determinate whether the signal will pass through the net or not.

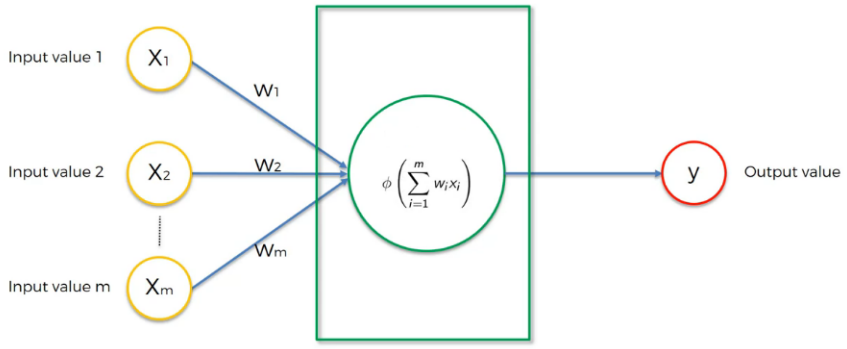

Figure 3.10: The activation function ϕ

The Activation Function

There are four predominant types of activation functions that it is possible to choose from:

- the **threshold** function: on the x axis there is the weighted sum of inputs while on the y axis there is the threshold that varies from 0 to 1. As figure 3.11 shows, if the value is less than zero the threshold function passes on zero, while if the value is more than zero then the function goes on 1.

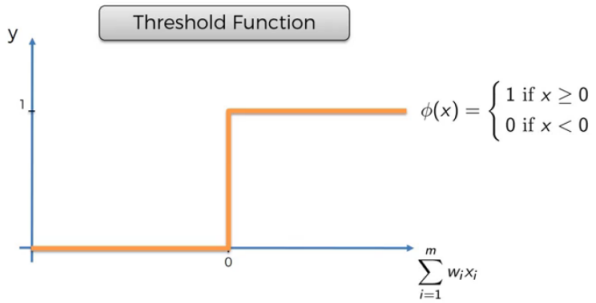

Figure 3.11: The Threshold function

- the **sigmoid** function: unlike the threshold function, the sigmoid is smooth and varies gradually as shown below.

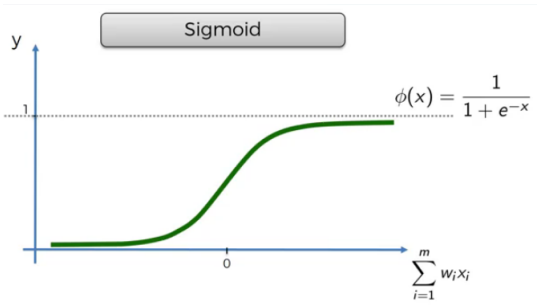

Figure 3.12: Sigmoid function

- the **rectifier** function: it is one of the most popular activation function in artificial neural network. It gradually progresses as follows

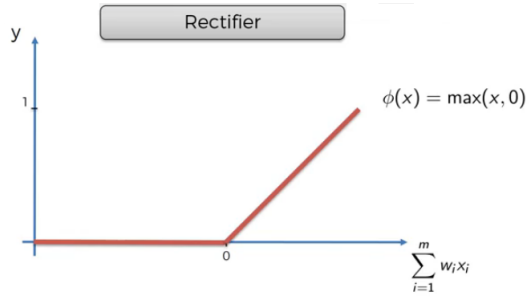

Figure 3.13: The Rectifier function

- the **Hyperbolic Tangent** function: it is similar to the Sigmoid function but in this case the threshold goes below -1 to 1 and this can be useful for some applications.

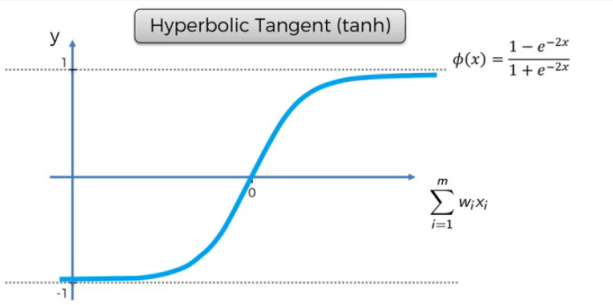

Figure 3.14: The Hyperbolic tangent function

Usually, a combination of two different activation functions may be used by the neurons as shown in figure 3.15 where at first a rectifier function is applied to the weighted sum in the hidden layer and then a sigmoid in the output layer returns the output value.

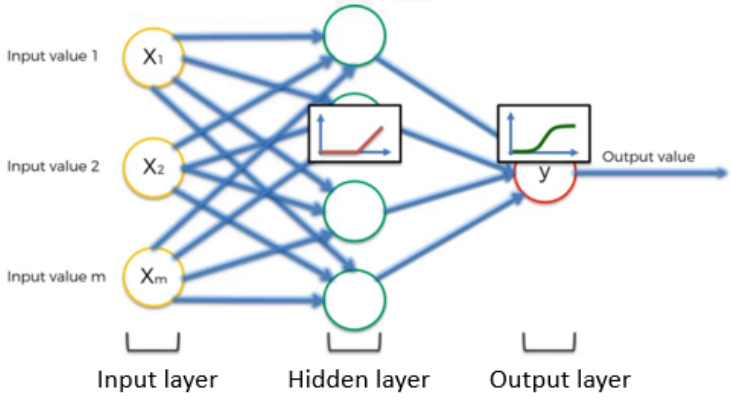

Figure 3.15: Activation function application

How Neural Networks learn

There two fundamentally different approaches to get a program to do what it is meant to do. One is **hard coded coding** where we actually tell the program specific rules and what outcomes we want – we account for all the possibilities and we guide the program throughout the whole way. On the other hand are **neural networks** where we crate a facility for the program to be able to understand what it needs to do on its

own – we basically provide inputs and what is the desired output and then we let the net figure everything out on its own. For instance, let assume we want to distinguish between cats and dogs. If we follow the first approach we should tell the program to look for specific features (look out for nose, ears and face) and we have condition – if the ears is pointy than it is a cat, if the ears are sloping down it may be a dog. On the other hand, for a neural network we need to code the architecture and then we point the neural network at a folder with images of cats and dogs which are already categorized. The neural network will on its own understand everything it needs to understand and then, once it is trained up, we can give it an unseen image of a cat or a dog and it will be able to understand what it is.

Figure 3.16 shows a single layer feed-forward neural network or **perceptron** which is a very basic neural network with one single hidden layer. The output value here is indicated as $\hat{y}$. The reason is that y stand for the actual value that we see in reality while $\hat{y}$ is the value predicted by the neural network.

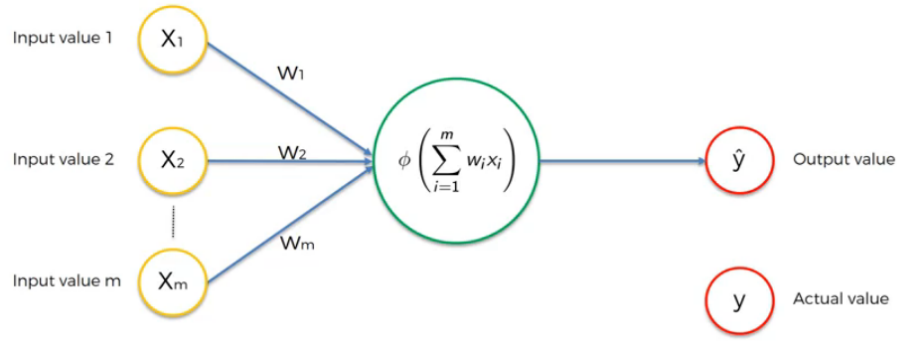

Figure 3.16: A single layer feed-forward neural network

Some input values are supplied to the perceptron, the activation function is applied and then we get the output. In order to be able to learn, the output value has to be compared to the actual one. After this step, we can get the **cost function** C which is calculated as

$$C = \frac{(\hat{y} - y)^2}{2}$$

and it tells what is the error in the neural net prediction. The goal is to minimize the cost function since the lower the cost function the closer $\hat{y}$ is to y. Once the output value and the actual value are compared, the information is fed back into the neural network as shown in figure 3.17 and weights get updated.
It is important to highlight that so far we have just taken into consideration one line of a dataset which provides for instance three input values to the net – we want to predict the chance to pass an exam for a student and we feed the net with the number of study hours, sleep hours and what they get at the quiz.

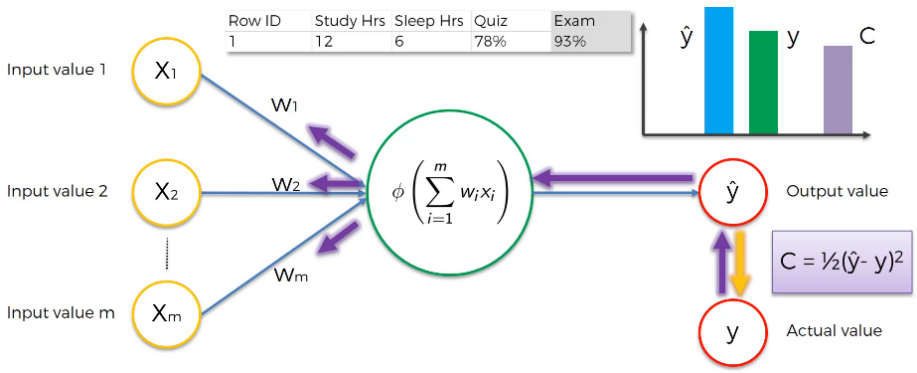

Figure 3.17: Cost function and backwards propagation for one line of the training data set

If there are multiple rows, for instance eight rows, the situation is the one shown in the figure 3.18 – the eight rows are all feed in the **same** neural network. It is called **epoch** when the net gets trained on the whole data set. By doing so we will get one predicted value $\hat{y}$ for each row that we have to compared to the actual values y. Now, we can calculated the cost function which is the sum of all the squared differences between $\hat{y}$ and y. After we get the whole cost function we propagate the information backwards and the weights get updated. The goal here is to minimize the cost function – once we get the minimum, that means we got the optimal weights for the data set we trained the net on and we can proceed with the testing phase. This all process is called **back propagation**.

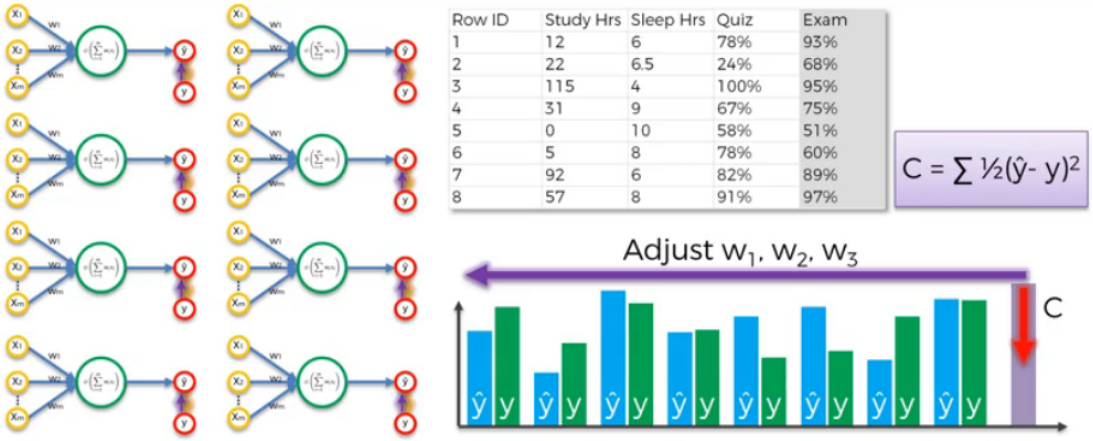

Figure 3.18: Cost function and backwards propagation for the entire training data set

Gradient Descent

To minimize the cost function, we could take lots of different possible weights and look which one look best.

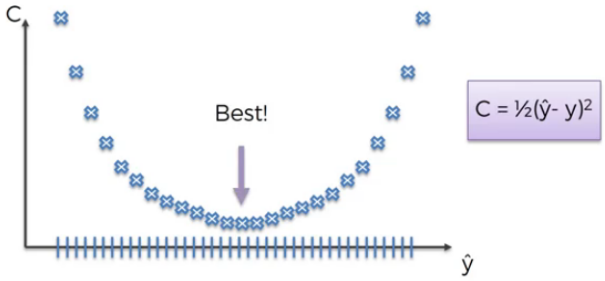

Figure 3.19: Minimizing the cost function

This approach is intuitive, but as soon as the number of weights increase we have to face the curse of dimensionality. The best way to explain this concept is to look at a practical example. Let build and run a neural network for a property valuation. The actual neural network (when not trained) looks like in figure – there are 25 synapses. If, for instance, there are 1000 of input combinations that means that there are

$$1000^{25} = 10^{75} combinations$$

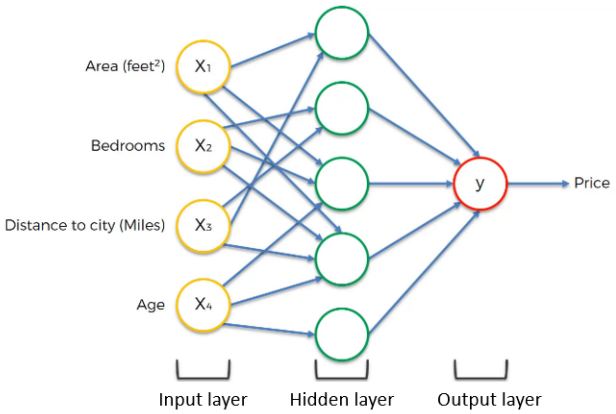

Figure 3.20: Understanding Gradient Descent – a neural network for property valuation

Now, let consider Sunway TaihuLight, world's fastest Super computer, that can operate at 93 PFLOPS where flops stands for floating operation per second – which is basically $93 \cdot 10^{15}$ operations per second. Let's say hypothetically that it can do one combination for our own network. To test out a single weight one single floating operation is ideally enough – that means that it would still require $10^{75}/(93 \cdot 10^{15}) = 1.08 \cdot 10^{58}$ seconds which is equal to $3.42 \cdot 10^{50}$ years which is longer than the universe has existed. It is obviously not going to work for the optimization process and a different approach is required. This approach is named **Gradient Descent**. As the name suggests, we need to differentiate to find out what the slope is until we get that it is equal to zero at the optimal point.

Stochastic Gradient Descend

Gradient Descent is a very efficient method to solve the optimization problem where we are trying to minimize the cost function. This method requires for the cost function to be convex so that it has one global minimum. If we choose a cost function which is not the square difference between $\hat{y}$ and y, we could find a local minimum of the cost function rather than the global one and therefore we do not have an optimized neural network. The solution here is **Stochastic Gradient Descent** which does not require for the cost function to be convex. To illustrate how these two approaches work, we consider again the example of the previous section about exam result prediction. **Normal gradient descent** takes all the rows of the dataset the net is trained on and plugs it in the same neural network every time. Once they are plugged in, the cost function is calculated and weights are adjusted. On the other hand, **stochastic gradient descend** takes the rows one by one, adjusts the weights and then moves onto the next row – it basically adjusts the weights after every single row rather than doing everything together and then adjusting weights.

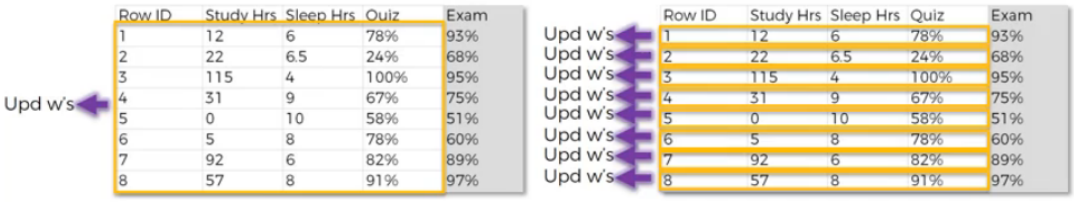

Figure 3.21: A practical comparison between Normal gradient descend and stochastic gradient descend

The stochastic gradient descent helps avoiding the problem of local minimum. The reason for that is that it can afford much higher fluctuations – since it is doing one iteration at a time, the fluctuations are much higher and it is much more likely to find the global minimum rather than the local minimum. Moreover, the stochastic gradient descent is faster than the normal one because it does not have to load up all

the data into memory. However, the main advantage of the normal gradient descent is that it is a deterministic algorithm rather than stochastic – as long as the staring weights are the same the results are going to be the same. In the stochastic scenario, the rows are picked at random and even if the starting weights are the same the iterations are different. There is also a method in between the two called **the mini-batch gradient descent** where batches of 5, 10 or 100 rows are run at the same time rather the one or the whole.

Back propagation

In previous sections, we illustrated that there is a process called forward propagation where information is entered into the input layer and then it is propagated forward to get the predicted value $\hat{y}$ which is then compared to the actual value. Then the errors are calculated and back propagated through the network in the opposite direction and that allows us to train the network by adjusting the weights. The back propagation is an advance algorithm driven by very sophisticated mathematics which allows weights to be adjusted **simultaneously** – by doing so, we can basically know which part of the error each of the weights is responsible for. This is the main principle the back propagation is built on.

Training the ANN with Stochastic Gradient Descend

In this section, we are going to wrap everything up with a step by step walk through of what happens in the training of a neural network:

1. Randomly initialize the weights to small numbers close to 0 (but not 0) that through the process of propagation are adjusted until the error is minimized.
2. Input the first observation of the dataset in the input layer where each feature is on input node.
3. Forward-propagation: from left to right, the neurons are activated in a way that the impact of each neuron's activation is limited by the weights - propagate the activation until getting the predicted result y.
4. Compare the predicted result to the actual result and measure the generated error.
5. Back-propagation: from right to left, the error is back-propagated. Update the weights according to how much they are responsible for the error. The learning rate decides by how much we update the weights.
6. Repeat steps 1 to 5 and update the weights after each observation (**reinforcement learning**). Or: Repeat steps 1 to 5 but update the weights only after a batch of observations (**batch learning**).
7. When the whole training set passed through the ANN, that makes an epoch. Redo more epochs

3.5.2 Convolutional Neural Networks

Another important class of deep neural networks is that of **Convolutional Neural Network**, most commonly applied to analyzing and classifying images. It basically takes an input image that goes to the net itself and it returns an output label with a certain confidence.

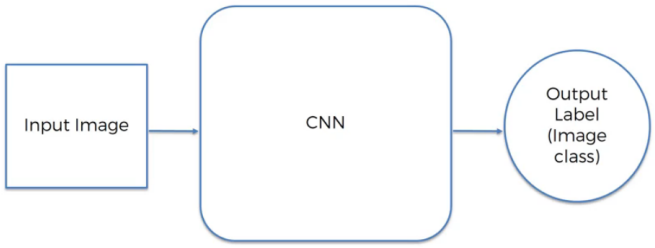

Figure 3.22: Convolutional Neural Network

To label an image, CNN looks for some features on which it has been previously trained. CNN recognizes features at a very basic level. Let's assume we have two images of two by two pixels, one is black and white and the other is colored. The computer sees the black and white image as a two dimensional array with every pixel having a value between 0 and 255 – 0 will be a completely black pixel, 255 a completely white one and between them there is the gray scale. On the other hand is the colored image which is a 3D array – red, green and blue layer where each one of those colors has its own intensity (each pixel has three values assigned to, each of them is between 0 and 255).

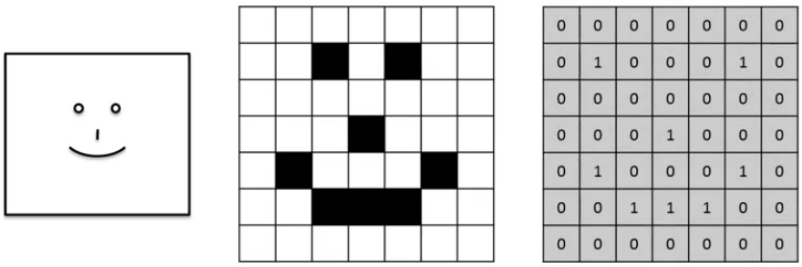

Figure 3.23: Visual and digital image information

The steps that CNN is applying to this information of images (pixel values) are:

1. Convolution
2. Max Pooling
3. Flattening
4. Full Connection

Convolution

A convolution is essentially the combined integration of two function and it shows how one function modifies the other.

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

To understand what is convolution in intuitive terms, we consider an input image and a *feature detector* (or kernel) which, in this example, is a three by three matrix. A convolutional operation is signified by a $\otimes$. It basically puts the feature detector on the input image (for instance it can cover the nine pixels at the top left corner), multiplies each value for the correspondent value and then adds up the result. The filter is then moved along the whole input matrix. The outcome of this operation is the **feature map** (also called convolved or activation map). It can be noticed that an important function of the convolution step is to make the input image smaller since it will be easier to process it. By convolving the kernel, some little information is lost since we have less values in the feature map but at the same time we do not have to look at each pixel in the image. The feature detector has a pattern on it, the highest number in the feature map is where that pattern matches up – number 4 in the feature map is where the feature detector matches perfectly with the input image. In conclusion, the feature map helps getting rid off the unnecessary things that even as humans we do not process.

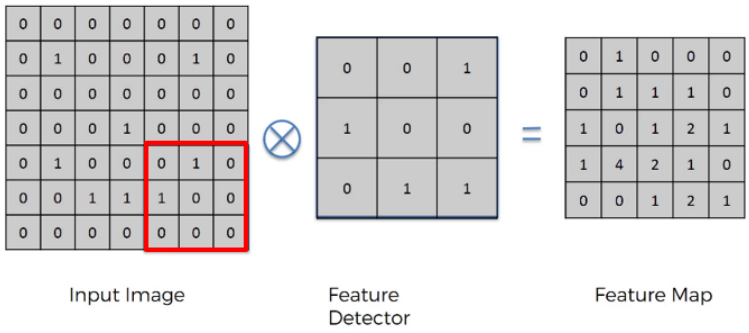

Figure 3.24: Convolution step – input image, feature detector and feature map

To preserve lots of information, it is possible to create multiple feature maps of the same image by using different filters (depending on the feature we want to detect). All the feature maps together form the **convolutional layer**.

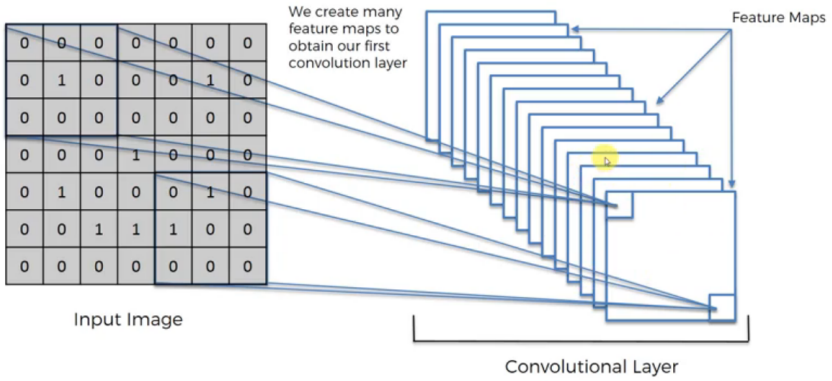

Figure 3.25: Feature maps and convolutional layer

Rectified Linear Unit

Rectified Linear Unit is an additional step performed on top of the convolution step. It basically consists in applying the rectifier function to the convolutional layer.

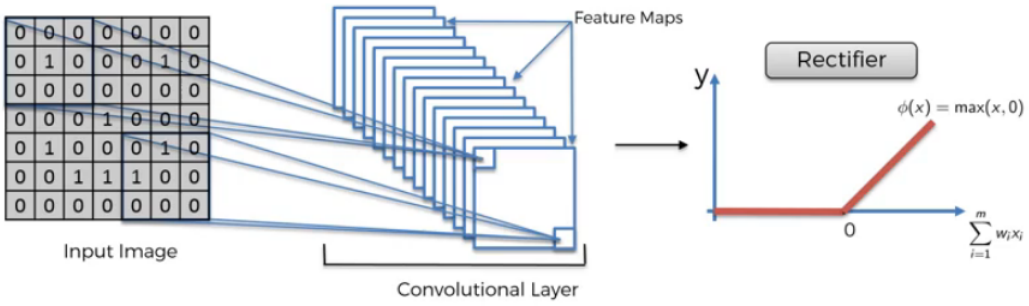

Figure 3.26: Rectified Linear Unit

Rectifier function is applied in order to increase non-linearity in the image. Images themselves are highly non-linear especially if there are different objects next to each other – the transition between adjacent pixels would be non-linear because of borders and different colors – but the reason why we want to increase non-linearity is because, when the feature detector is applied, there is the risk we might create something linear which do not help our network to distinguish between different patterns.

Max Pooling

We illustrated that the neural network is looking for features in order to label an image. These features may look different depending on the image that we are considering – they may appear rotated, positioned in different part of the image, under different light conditions and so on. That is the reason why the network have to be **spatial invariant** which means that it does not care where the features are, if they are tilted or different in texture with respect to the ones it has been trained on. So if a feature is a bit distorted, the neural network has to have some level of flexibility to be able to still find that feature. This is what **pooling** is all about.

To understand how pooling works, we consider a feature map – that means that convolution has already been performed – and we apply **max pooling** to it. To perform max pooling, a two by two box is taken at the top left corner of the feature map and just the maximum value within that box is recorder in the **Pooled Feature Map** while disregarding the other three. Then the box is moved by a stride, for instance two, and the operation is repeated until all the "surface" of the feature map is covered.

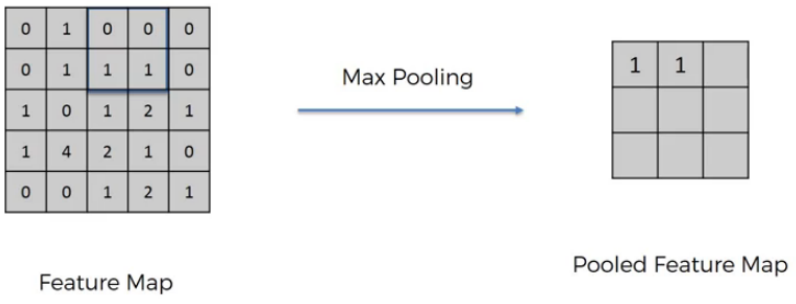

Figure 3.27: Max pooling

By looking at the pooled feature map, we can observe that:

1. the features are preserved. We have seen in the paragraph about convolution that the highest number in the feature map is where we find the closest similarity to a feature and this number is kept while pooling.

2. we are account for any possible spatial or textural invariance. By pooling the features we are getting rid of 75 % of information which, however, is not relevant. In fact, because we are taking the maximum of the pixels, we are already accounting for any distortion – the pooled feature is going to be the same even if the feature in the feature map is found at different location.

3. the size of the feature map is reduced by 75 % which it really helps in terms of processing.

4. the number of parameters is reduced of 75 % and therefore we are preventing over-fitting

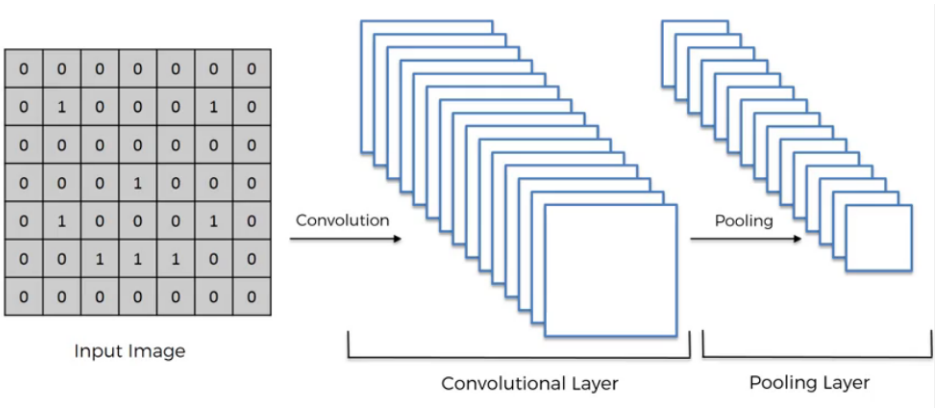

Figure 3.28: Convolutional and pooled layer

Flattening

We have seen that the result of applying convolution and max pooling is the pooled feature map which is basically a matrix of a given dimension. The next step that has to be performed is **flattening**, that as the name itself suggests, consists of flattening the pooled feature map into a column – it takes the number row by row and puts them into a column. The reason for this is because the column is later input into an artificial neural network for further processing. Figure 3.29 shows the flattening process in the presence of a pooling layer. All the feature maps are flattened into one long single column sequentially which represent one huge vector of input for an artificial neural network.

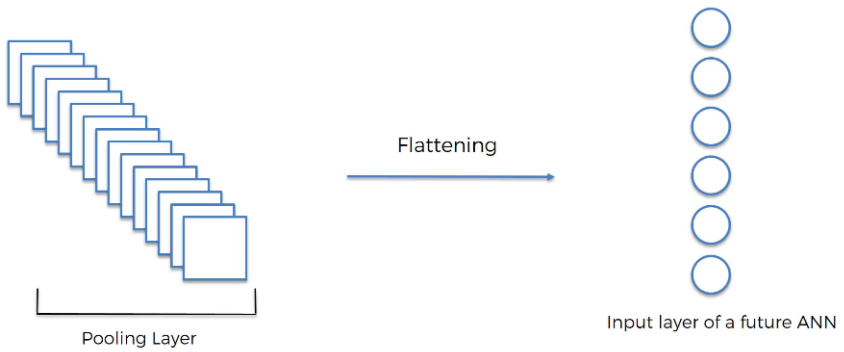

Figure 3.29: The flattening step

Full Connection

In this step, a whole artificial neural network is added to the convolutional neural network as shown in figure 3.28. There is an **input layer** which is the result of the flattening process describe above, one **fully connected layer** which is similar to what we previously called hidden layer and out **output layer**. Here, the main purpose of the artificial neural network is to combine the features extracted by the convolutional neural network into more attributes that can predict the class more efficiently. In fact, the flattening output vector has already some features encoded in the numbers which could already predict what class we are looking at – whether is an helicopter or an airplane. But at the same time we know that artificial neural networks which are designed for dealing with attributes can combine attributes together to even further optimize the classification algorithm.

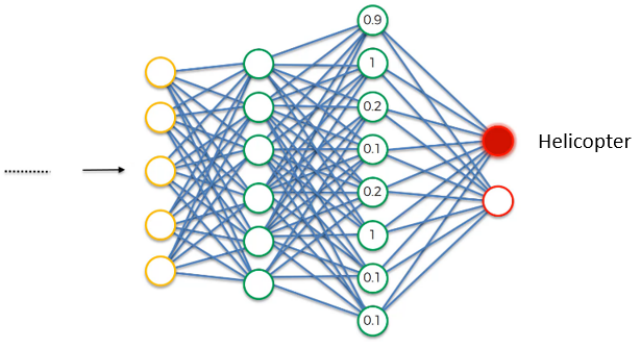

Figure 3.30: Full connection

Similarly to what we have seen for artificial neural network, after the information propagates through the net a lost type of function is calculated which tells us how well our network is performing. Once the error is calculated it is back propagated through the network and some things are adjusted to optimize the performance. The things that are adjusted are weights and feature detectors – we know that we are looking for feature, but what if we are looking for the wrong features? For this reason we are adjusted feature detectors.
It is relevant to highlight that in figure 3.30 there are two outputs. In fact, when the neural network is meant to perform classification, one output for each of the class the net is trained on is required – the only exception is when there are only two classes and one binary output is enough (for instance, 1 stands for helicopter and 0 stand for airplane). To better understand how these two outputs work, let's start with the top output neuron which we assume that predicts for helicopters. First, we need to understand what weights to assign to all the "synapses" that connect to the helicopter node so that we know which neurons are actually important. Let's assume hypothetically that in the previous fully connected layer we got some numbers between 0 and 1 where 1 means that the neuron was very confident that it found a certain feature and 0 that it was not. In figure 3.30 , the second neurons has a value of 1 which is passed to both the Helicopter and the Airplane output neuron but depending on how relevant that feature is for classifying the target, the weight is going to be different – and through epochs and iterations, the optimal weight is decided by the output neuron itself that knows its own class and can thus understand which features are relevant or not.

3.6 Object detection with Machine Learning

In section 3.2, we listed the main algorithms for object detection that are based on a machine learning approach. Among them, there is the **Viola–Jones object detection framework** which is considered the first competitive real-time object detection system. This algorithm uses an ensemble approach, instead of looking at the whole image. It basically uses different classifiers, each looking at a different portion of the image. The main idea is that each individual classifier is weaker (less accurate, produces more false positives, etc) than the final classifier because it is taking less information. But when the results from each classifier are combined, however, they produce a stronger classifier.

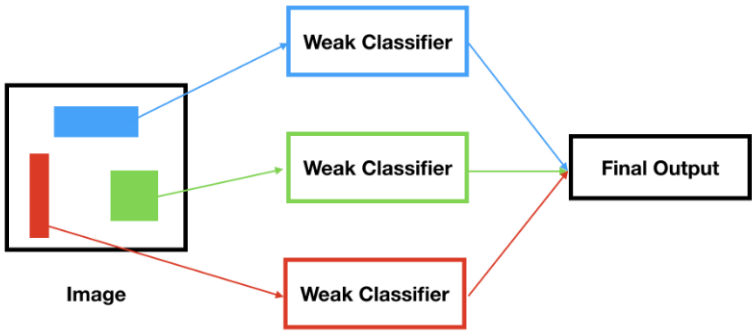

Figure 3.31: The Viole-Jones algorithm ensemble approach

Although it can be trained to detect a variety of object classes, it was first applied to face detection problems (not recognition – the goal is to distinguish faces from non-faces). In order to perform the detection, the algorithm performs the **Haar Feature Selection**. In figure 3.31, we can see that the features sought by the framework are the sum of image pixels within rectangular areas. Figure 3.32 illustrates some Haar Feature examples: the first two are *edge features*, used to detect edges. The third is a *line feature*, while the fourth is a *four rectangle feature*, most likely used to detected a slanted line. For instance the second filter looks for an area that is dark on top and brighter underneath. In face detection, for instance, it can be used to detect eyes since the eye region is darker than the upper-cheeks.

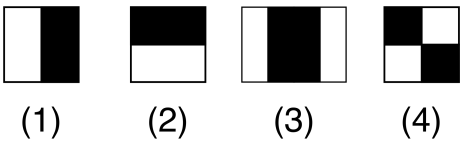

Figure 3.32: Haar Features examples

Without entering deeply into mathematics , the Viola-Jones results to be a robust and real time algorithms – very high detection rate (true-positive rate) and very low false-positive rate – but however it always required full view frontal upright faces (or objects in general). Thus in order to be detected, the entire face must point towards the camera and should not be tilted to either side. These are the main constrains that could diminish the algorithm utility somewhat.

Another feature descriptor used in computer vision and image processing for the purpose of object detection is the **histogram of oriented gradients (HOG)**. HOG is used to characterize objects on the basis of their shapes by calculating histogram (occurrences) of each gradient orientation. The core idea behind the histogram of oriented gradients descriptor is that local object appearance and shape within an image

can be described by the distribution of intensity gradients or edge directions as shown in figure 3.33.

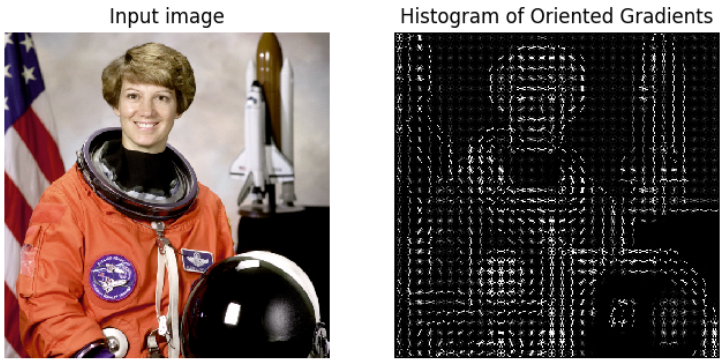

Figure 3.33: Histogram of oriented gradient

Once HOG (or Viole-Jones) has extract the features within the image, a linear **Support Vector Machines (SVM)** can be use to perform the classification of the combined features. SVM can be define as machine learning models that performs supervised learning. Given a set of training datasets, each marked as belonging to one or the other of two categories, a SVM builds a model that assigns new examples to one category or the other. The objective of the support vector machine algorithm is to find a **hyperplane** in an N-dimensional space (where N is the number of features) that distinctly classifies the data points. To separate the two classes of data points, there are many possible hyperplanes that could be chosen. The objective is to find a plane that has the maximum margin, i.e the maximum distance between data points of both classes. Maximizing the margin distance provides some reinforcement so that future data points can be classified with more confidence.

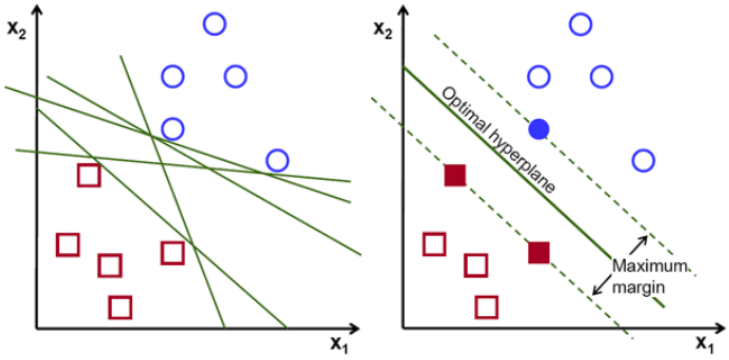

Figure 3.34: Possible Hyperplanes

Data points falling on either side of the hyperplane can be attributed to different classes. Also, the dimension of the hyperplane depends upon the number of features. If the number of input features is 2, then the hyperplane is just a line. If the number of input features is 3, then the hyperplane becomes a two-dimensional plane. It becomes difficult to imagine when the number of features exceeds 3.
Although SVMs have good generalization performance, they have some drawbacks. The most serious problem is the high algorithmic complexity and extensive memory requirements. Speed and size is another problem both in training and testing. It can be demonstrated that in terms of running time, SVMs are slower than other neural networks. Given these constrains, for the purposes of this projects a deep learning based approach will be adopted to perform the object detection step.

3.7 Object detection with Deep Learning

After the rise of deep learning, the aim was to replace HOG based classifiers with a more accurate convolutional neural network based classifier. However, CNNs are mainly for image classification which means that a typical CNN can tell us the class of the objects within the image but not where they are located. However, it would be possible to divide the input image into various regions and consider each region as a separate image and then pass all these regions to the net that classifies them into classes. But by doing so, the CNN would result extremely slowly and computationally expensive. To overcome this limit, the **Regional Convolution Neural Networks (R-CNN)** were introduced. R-CNN solves the problem of object detection (classification + localization) by using an object proposal algorithm called *selective search* that uses local cues like texture, intensity, color etc to generate all the possible locations of the object. In other words, the network does not look at the entire image, but only at the parts of the images which have an higher chance to contain an object. Selective search reduces the number of bounding boxes that are fed to a CNN classifier to close to 2000 region proposals.

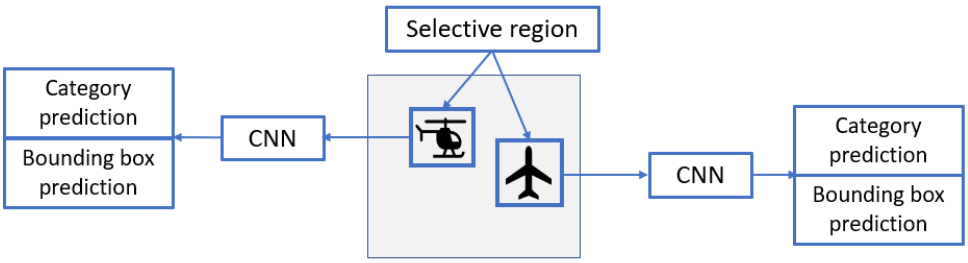

Figure 3.35: Regional Convolutional Neural Network

Still, R-CNN can be really slow. The main performance bottleneck of an R-CNN model is the need to independently extract features for each proposed region. Running CNN on 2000 region proposals generated by selective search may take a lot of time. As these regions have a high degree of overlap, independent feature extraction results in a high volume of repetitive computations. Moreover, the selective search algorithm is a fixed algorithm. Therefore, no learning is happening at that stage and this could lead to the generation of bad candidate region proposals. The same author of the previous paper(R-CNN) solved some of the drawbacks of R-CNN by building a faster object detection algorithm that was called **Fast R-CNN** (figure 3.36).

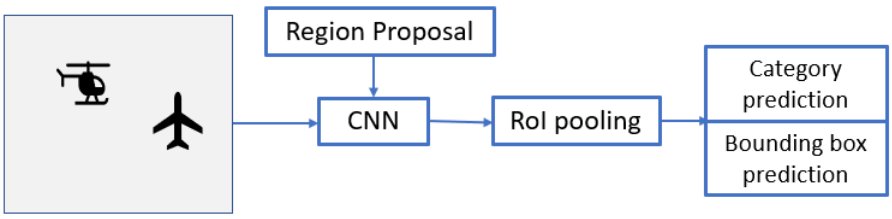

Figure 3.36: Fast-Regional Convolutional Neural Network

Fast R-CNN improves on the R-CNN by only performing CNN forward computation on the image as a whole. The approach is similar to the R-CNN algorithm. But, instead of feeding the region proposals to the CNN, we feed the input image to the CNN to generate a convolutional feature map. From the convolutional feature map, we identify the region of proposals and warp them into squares and by using a *Region*

of Interest (RoI) pooling layer we reshape them into a fixed size so that it can be fed into a fully connected layer. From the RoI feature vector, we can predict the class of the proposed region.

Both of the above algorithms (R-CNN and Fast R-CNN) uses selective search to find out the region proposals. Selective search is a slow and time-consuming process affecting the performance of the network. Therefore **Faster R-CNN** were introduced that eliminate the selective search algorithm and let the network learn the region proposals. Similar to Fast R-CNN, the image is provided as an input to a convolutional network which provides a convolutional feature map. Instead of using selective search algorithm on the feature map to identify the region proposals, a separate network is used to predict the region proposals. The predicted region proposals are then reshaped using a RoI pooling layer which is then used to classify the image within the proposed region.

	R-CNN	Fast R-CNN	Faster R-CNN
Test Time per Image	50 Seconds	2 Seconds	0.2 Seconds

Figure 3.37: A comparison between R-CNN, Fast R-CNN and Faster R-CNN

All of the previous object detection algorithms use regions to localize the object within the image. The network does not look at the complete image but instead, parts of the image which have high probabilities of containing the object. To speed up the performances of the object detection, **YOLO (You Only Look Once)** algorithm has been introduced. As the name itself suggest, YOLO takes the entire image in a single instance and predicts the bounding box coordinates and class probability for these boxes – this principle allows the algorithm to process up to 45 frames per second. Here follows a graphical comparison between the algorithms illustrated above in terms of speed and accuracy.

Figure 3.38: A comparison between Fast R-CNN, Faster R-CNN and YOLO

We can observe that even if the Faster R-CNNs have the highest accuracy, YOLO appears to be the fastest among the algorithms. Since speed is the primary requirement of our project, YOLO is the object detection model that we are going to elect for our purposes (moreover the difference is accuracy between Faster R-CNN and YOLO is minimal).

3.7.1 Introducing YOLO algorithm – a practical example

Now that we have briefly understand the core idea behind YOLO, let's describe how it actually works [4]. To perform object detection, YOLO:

1. first takes an input image – for example, a 100 x 100 pixels frontal view of two cars on the street (here the bounding boxes are designed just to visualize the two cars).

Figure 3.39: Input image for YOLO

2. then divides the input image into a **S x S** grid – say for instance a 3 x 3 grid.

Figure 3.40: Dividing the input image into grids

3. applies image classification and localization on each grid and predicts the bounding boxes and the corresponding class probabilities for objects (if any are found). Let assume there are a total of 3 classes which we want the objects to be classified into (for instance pedestrian, car, and motorcycle respectively). So, for each grid cell, we have an eight dimensional vector y:

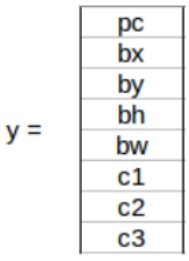

Figure 3.41: Predicting bounding boxes and class probabilities

where

- p_c defines whether an object is present in the grid or not – it is the probability.
- b_x, b_y, b_h, b_w specify the bounding box's normalized coordinates if there is an object. Depending on the implementation, b_x and b_y may be the coordinates of the left top corner while b_h and b_w are the height and the width of the bounding box.
- c_1 c_2 c_3 represent the classes. So, if the object is a car, c_2 will be 1 and c_1 and c_3 will be 0, and so on.

Let's consider the first grid from the above example. Since there is no object in this grid, p_c will be zero and the y label for this grid will be empty. But if we take a grid with a car in it, the outcome would be different.

Figure 3.42: The grid containing a car

Before writing the y label for this grid, it is important to first understand how YOLO decides whether there actually is an object in the grid. In the above image, there are two objects (two cars), so YOLO takes the mid-point of these two objects that are then assigned to the grid which contains the mid-point of these objects. Even if an object occupies more than one grid, it will only be assigned to a single grid in which its mid-point is located. The y label for the center left grid with the car will be:

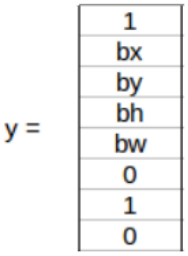

Figure 3.43: The eight dimensional vector for one detection

Since the grid contains an object among the ones the algorithms has been trained on, p_c will be equal to 1. b_x , b_y, b_w, b_h will be calculated relative to the particular grid cell we are dealing with. Since car is the second class, $c_2 = 1$ and c_1 and $c_3 = 0$. So, for each of the 9 grids, we will have an eight dimensional output vector. This output will have a shape of 3 x 3 x 8.

As we have seen, both forward and backward propagation will be run to train the model. During the testing phase, we pass an image to the model and run forward propagation until we get an output y. In order simplify things, a 3 x 3 grid has been considered, but generally in real-world implementation larger grids (up to 19 x 19) are employed. By doing so, the chances of multiple objects appearing in the same grid cell will be reduced.

Intersection over Union and Non-Max Suppression

In order to decide whether the predicted bounding box is giving a good outcome (or a bad one), the algorithm performs **Intersection over Union** (IoU) and **Non-Max Suppression**. Consider the actual and predicted bounding boxes for a car as shown below where the red box is the actual bounding box and the blue box is the predicted

one. IoU will calculate the area of the intersection over union of these two boxes. That area will be the yellow one:

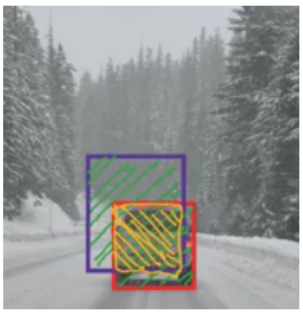

Figure 3.44: The Intersection over Union

Usually, if IoU is greater than 0.5, the prediction is considered to be good enough. 0.5 is an arbitrary threshold and it can be changed according to the specific problem. Intuitively, the higher the threshold, the better the predictions become.

The other principle that improves the output of YOLO significantly is Non-Max Suppression. One of the most common problems with object detection algorithms is that rather than detecting an object just once, they might detect it multiple times as figure 3.45 illustrates, where the cars are identified more than once.

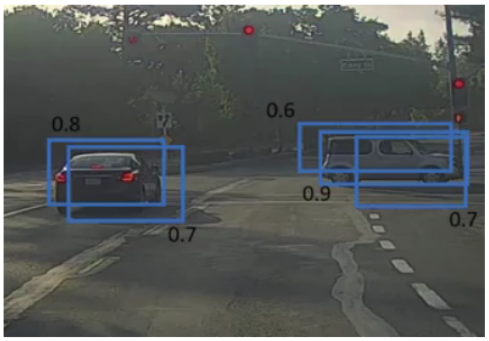

Figure 3.45: Multiple detection of the same object

The Non-Max Suppression technique performs the following steps:

1. it first looks at the probabilities associated with each detection and takes the largest one. In the above image, 0.9 is the highest probability, so the box with 0.9 probability will be selected first.

2. it looks at all the other boxes in the image. The boxes which have high IoU with the current box are suppressed. So, the boxes with 0.6 and 0.7 probabilities will be suppressed in the example.

3. after the boxes have been suppressed, it selects the next box from all the boxes with the highest probability, which is 0.8 in our case.

4. again it will look at the IoU of this box with the remaining boxes and compress the boxes with a high IoU.

5. These steps are repeated until all the boxes have either been selected or compressed and we get the final bounding boxes.

Anchor Boxes

We have seen that each grid can only identify one object. But what if there are multiple objects in a single grid? That can so often be the case in reality. This leads us to the concept of **anchor boxes**. Consider the following image, divided into a 3 x 3 grid. In the above example, the midpoint of both the objects lies in the same grid. This is how the actual bounding boxes for the objects will be:

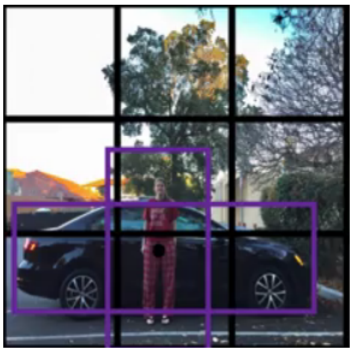

Figure 3.46: Anchor Boxes

We will only be getting one of the two boxes, either for the car or for the person. But if we use anchor boxes, we might be able to output both boxes. To achieve that, we first define two different shapes called anchor boxes or anchor box shapes. Now, for each grid, instead of having one output, we will have two outputs. We can always increase the number of anchor boxes as well.

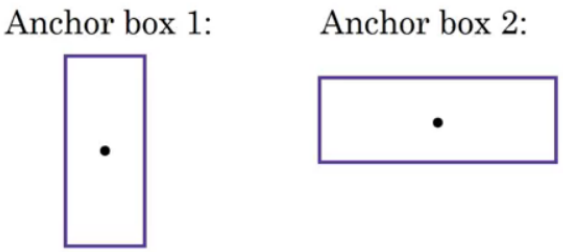

Figure 3.47: Example of Anchor Boxes

The objects are assigned to the anchor boxes based on the similarity of the bounding boxes and the anchor box shape. Since the shape of anchor box 1 is similar to the bounding box for the person, the latter will be assigned to anchor box 1 and the car will be assigned to anchor box 2. The output in this case, instead of being 3 x 3 x 8 (using a 3 x 3 grid and 3 classes), will be 3 x 3 x 16 (since we are using 2 anchors). So, for each grid, we can detect two or more objects based on the number of anchors. In conclusion, the predictions are encoded in a

$$S \times S \times (B * 5 + C) \tag{3.1}$$

where S is the number of grids, B the number of bounding boxes that each cell can predict and C the number of classes.

3.7.2 YOLO Design

YOLO was first implemented as a convolutional neural network [5] and tested on the PASCAL VOC data sets [6] that has 20 labeled classes. For evaluating YOLO on PASCAL VOC data sets, the authors used $S = 7$ and $B = 2$ so that the final prediction was a $7 \times 7 \times 30$ tensor according to equation 3.1. The network has a 448×448 input resolution and it is made of 24 convolutional layers and max pooling layers, followed by 2 fully connected layers so that a $7 \times 7 \times 30$ tensor could be predicted. The full network is shown in figure 3.48.

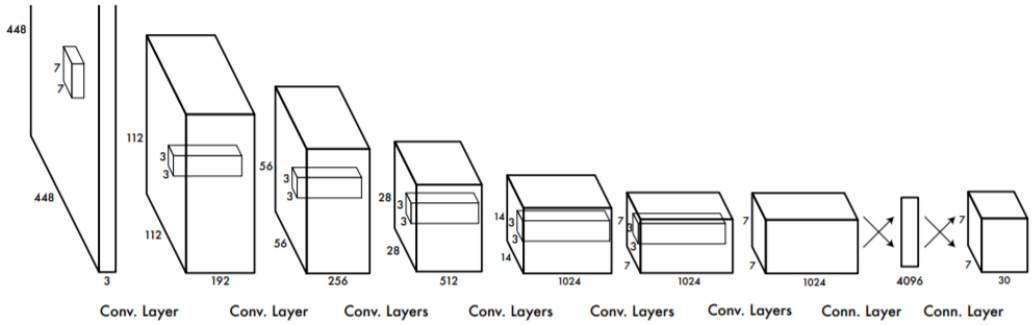

Figure 3.48: YOLO Network

The initial convolutional layers of the network extract features from the image while the fully connected layers predict the output probabilities and bounding boxes coordinates.

Name	Filters	Output Dimension
Conv 1	7 x 7 x 64, stride 2	224 x 224 x 64
Max Pool 1	2 x 2, stride 2	112 x 112 x 64
Conv 2	3 x 3 x 192	112 x 112 x 192
Max Pool 2	2 x 2, stride 2	56 x 56 x 192
Conv 3	1 x 1 x 128	56 x 56 x 128
Conv 4	3 x 3 x 256	56 x 56 x 256
Conv 5	1 x 1 x 256	56 x 56 x 256
Conv 6	1 x 1 x 512	56 x 56 x 512
Max Pool 3	2 x 2, stride 2	28 x 28 x 512
Conv 7	1 x 1 x 256	28 x 28 x 256
Conv 8	3 x 3 x 512	28 x 28 x 512
Conv 9	1 x 1 x 256	28 x 28 x 256
Conv 10	3 x 3 x 512	28 x 28 x 512
Conv 11	1 x 1 x 256	28 x 28 x 256
Conv 12	3 x 3 x 512	28 x 28 x 512
Conv 13	1 x 1 x 256	28 x 28 x 256
Conv 14	3 x 3 x 512	28 x 28 x 512
Conv 15	1 x 1 x 512	28 x 28 x 512
Conv 16	3 x 3 x 1024	28 x 28 x 1024
Max Pool 4	2 x 2, stride 2	14 x 14 x 1024
Conv 17	1 x 1 x 512	14 x 14 x 512
Conv 18	3 x 3 x 1024	14 x 14 x 1024
Conv 19	1 x 1 x 512	14 x 14 x 512
Conv 20	3 x 3 x 1024	28 x 28 x 512
Conv 21	3 x 3 x 1024	14 x 14 x 1024
Conv 22	3 x 3 x 1024, stride 2	14 x 14 x 1024
Conv 23	3 x 3 x 1024	7 x 7 x 1024
Conv 24	3 x 3 x 1024	7 x 7 x 1024
FC 1	–	4096
FC 2	–	7 x 7 x 30 (1470)

Table 3.1: YOLO layers

Use of this network with a different grid size or different number of classes might require tuning of the layer dimensions.

In its first version, YOLO imposes strong spatial constraints on bounding box predictions since each grid cell only predicts two boxes and can only have one class. This spatial constraint limits the number of nearby objects that the model can predict. And moreover the model struggles with small objects that appear in groups, such as flocks of birds. For these reasons, various improvements were proposed leading to a new version of the model, **YOLOv2**. The new model divides the image into 13×13 grid cells which is smaller compared to its previous version. This enables the YOLOv2 to identify or localize the smaller objects in the image. Moreover, YOLOv2 uses **Darknet-19** architecture with 19 convolutional layers (instead of 24) and 5 max pooling layers and a softmax layer for classification objects. Darknet is an open source neural network framework written in C language and CUDA that supports CPU and GPU computation and it is used as the framework for training YOLO, meaning it sets the architecture of the network itself.

Even if already some relevant enhancements were introduced, YOLOv2 was still improved in an update version which is called **YOLOv3**. The most salient feature of the third version is that it performs detection at three different scales which are precisely given by downsampling the dimensions of the input image by a factor of 32, 16 and 8 respectively. This allows the network to learn and predict the objects from various input dimensions with accuracy and thus dealing better with different scales. Compared to its predecessor, YOLO v3 is much deeper. First, YOLOv3 uses a variant of Darknet, which originally has 53 layer network. For the task of detection, 53 more layers are stacked onto it, giving us a 106 layers. As a consequence, the last version of the model results to be slower compared to the v2 as shown in table 3.2.

Model	Backbone	FPS
YOLOv2	DarkNet-19	40
YOLOv3	DarkNet-106	20

Table 3.2: Comparison of backbones

3.7.3 YOLO training

YOLOv3 was trained on COCO data sets [7] which is an excellent object detection data set with 80 classes, 80,000 training images and 40,000 validation images. Even if the amount of training images is consistent, the model will be able to detect the 80 classes only under specific conditions which are the ones that characterize the training data sets. Let's take an example. The model is trained to detect cars and it actually detects cars pretty efficiently as figure 3.49 shows.

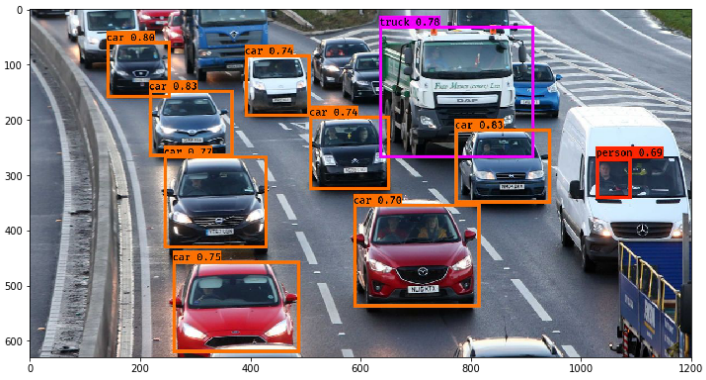

Figure 3.49: An example of YOLO detection

But if we test the model on a image with cars taken from overhead, the outcome would be the following (figure 3.50). This practical example highlights how data sets affect the "knowledge" of the model. Same features of same object may lead to bad outcomes if we test the model on **prospective**, **scale**, **lighting conditions** that are different from the ones it is trained on.

Figure 3.50: An example of YOLO wrong prediction

In the contest of this work, if we used COCO's weights to perform detection our tracking system would fail. Here comes the need to train the model on a custom data set from which we can extract the features of our interest.

Training and testing data sets

Deep learning projects depend heavily on data, since without data it is impossible for AI to learn. Data sets, which are collection of data, are the most crucial aspect that make the training possible. As shown in the previous section, if the training data set is not meaningful enough, the entire AI project will fail.
During the entire deep learning project development, we always rely on data. From training to testing , we basically use two type of data set: the **training set** and the **testing set**.

1. The **training data set** is the set of data through which the model learn how to process information and makes up the majority of the total data, around 70%.

Since our project deals with image processing and computer vision, our training test will be composed of large numbers of images. An iterative training on each of those images will eventually lead the model to recognize features, shapes and subjects of as cars or trucks from overhead. The training data set will include both input and desired output (the so called *ground truth*). Later it will be illustrated how our training set will be structured.

2. The **testing data set** is the set of data to evaluate how well the model was trained with the training set. It is important to highlight, that the testing set has to be different from the training set, otherwise the model will know in advance the expected output. Testing sets represents around the 20% of the entire data. Also the testing set comes together with its ground truth.

Given so, the first requirement for our custom data set is that it has to include images of cars or trucks taken from overhead – basically, aerial images either taken from aircraft or unmanned aerial vehicles . However, depending on the altitude of the photography platform and on the sensor, the cars may appear smaller or bigger. So, also the size in pixels of the target has to be checked.
Here, some operating requirements come into play:

1. the **working distance**. The tracking system is meant to be mount on the P92 SmartBay aircraft whose cruising altitude is around 1000 feet ($\simeq$ 300 m).
If the angle between the horizontal plane and the line of sight measured on the vertical plane is 60°, the situation can be sketched as shown in figure 3.51

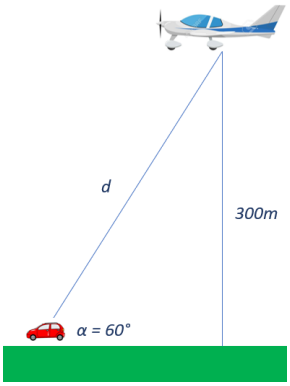

Figure 3.51: Cruising altitude and elevation angle

where the distance d in line of sight is

$$d = \frac{Altitude}{\sin(\alpha)}$$

2. the **sensor**. The camera employed in the first prototype of the model is planned to be the *Black Magic Studio Camera 4K* with the *Panasonic H-PS14042 Lenses*. Some camera specifications are summarized in table 3.3.

Effective Sensor Size	13.056mm × 7.344mm		
Focal Length	14mm	25mm	42mm
Shooting Resolutions	3840 × 2160	1920 × 1080	1280 × 720

Table 3.3: Black Magic Studio Camera 4K and lenses specifications

By using the equation 2.1, we can compute the angles of view of the camera for each focal length (table 3.4. Now, since we know the distance d and the HFOV of the camera we can get the corresponding width of the image in meters as

$$W_{imm} = 2 * d * \tan\frac{\theta_h}{2}$$

Focal Length	14mm	25mm	42mm
HFOV	50°	29°	17°
W_{imm}	323m	180m	107m

Table 3.4: Focal length, HFOV and Image width

Moreover, given the shooting resolution and the sensor size, we can derive the dimension of the pixel in mm p as

$$p = \frac{Sensor\,size}{resolution}$$

Considering the lower resolution and the size of the CCDW reported in the table the pixel dimension p will be 0.0102 mm.

Another parameter that has to be taken into account is the effective size of the target which is meant to be detected. Common cars usually have a length that varies from 3.5m up to 5m. How many pixels these values correspond to? To answer this question, we first need to fix some parameter which are:

- the **focal length** f which is set the medium value of 25mm;
- the **altitude** a (300m);
- the **effective size** of the target that we set to a medium value of 4m.
- the **resolution** r. In order to speed the image processing step and reduce the computational cost, we set the resolution at 720p.

The resulting size in pixel L_p of our 4m car can be derive as

$$L_p = L * \frac{r}{W_{imm}} \simeq 28\,pixels$$

Given these requisites, we looked for training and testing sets containing images with cars of this pixel size. One option was to build the data sets on our own but taking, classifying and labeling the a large amount of images would take a lot of time. For these reasons, the idea of producing data was initially set aside.
While searching for meaningful data sets that could satisfy our requirements, the **Car Overhead With Contest** (COWC) [8] was found. The data set and all related material is made *publically* available and can be freely downloaded. This large diverse set of cars from overhead images was presented to count and localize cars instances in an image and results to be pretty useful for training a deep learner that classifies, detects and counts cars. A problem encountered when trying to create a system for these purposes is the lack of large public data sets. For instance $OIRDS$ [9] has only 180 unique cars and $VEDAI$ [10] has 2950. Both of these data sets appear to be limited by not only the number of unique objects, but also because they tend to cover the same region and use the same sensor. $COWC$ contains a larger number of unique cars (32,716) from six different image sets each covering a different geographical region and moreover are taken by different sensor.
The regions covered are:

- $Toronto$ – $Canada$;
- $Selwyn$ – $New\,Zealand$;
- $Potsdam$ and $Vaihingen$ – $Germany$;
- $Columbus$ and $Utah$ – $United\,States$;

Two of the sets (Vaihingen, Columbus) are grayscale. The other four are in RGB color.

The set is also thought to be difficult as it is evident in figure 3.52, where many occlusions are present.

Figure 3.52: A *tricky* image from COWC data set

It contains 58,247 usable negative targets that are easy to confuse with cars. Examples of these are boats, trailers, bushes and skylights. To compensate for the added difficulty, context is included around targets. The general idea is to allow a deep neural network to understand and determine the weight between context and appearance such that something that looks very much like a car is detected even if it is in an unusual place.

As regards some data set details, all sets of images are sized 256×256 and the resolution is standardized to $15\,cm$. This makes cars range in size from 24 to 48 pixels. A patch is considered to contain a car if it appears in a central 48×48 region (the largest expected car length). Any car outside this central region is considered context. An edge margin of **32 pixels** was grayed out in each patch. All cars in the annotated images have a dot placed on their center. Cars that have occlusions are included so long as the annotator is reasonably sure the item is a car. Large trucks are completely omitted since it can be unclear when something stops being a light vehicle and starts to become a truck. Vans and pickups are included as cars even if they are large. All boats, trailers and construction vehicles are always added as negatives.
Eventually, each car of the data set is labeled as either:

- Sedan;
- Pick-up;
- Others;
- Unknown.

As we can see in figure 3.53, each class of car is labeled with a bounding box of a different color according to the RGB scale: *Sedan*'s are **red**, *Pickup*'s are **green**, *Others*'s are **blue** and $Unknown$'s which is not shown in the figure are **black**.

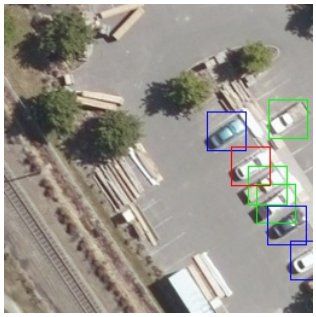

Figure 3.53: A *labeled* image from COWC data set

Along with the original image and the labeled image comes a $file.txt$ which contains a number of rows equal to the number of labeled cars within the image as figure 3.54

shows. Each row has the same structured:

$$< object - class >, < x >, < y >, < width >, < height >$$

where

- the first element stands for the *object class* and it is an integer number from 1 to the number of classes;
- $< x >, < y >$ are float values that represent the coordinates of the center of the bounding box relative to the width and the height of image – they varies from 0.0 to 1.0;
- $< width >, < height >$ are float values that represent the width and the height of the bounding boxes (which are squared) relative to the to the width and the height of image. Since all images and bounding boxes are equal sized, these two value are always 0.125.

```
2 0.91796875 0.37109375 0.125 0.125
3 0.71875    0.41015625 0.125 0.125
2 0.8515625  0.5859375  0.125 0.125
2 0.87890625 0.64453125 0.125 0.125
3 0.9140625  0.71484375 0.125 0.125
3 0.98828125 0.828125   0.125 0.125
1 0.796875   0.5234375  0.125 0.125
```

Figure 3.54: Bounding box information within the file.txt

Through this overview, it is evident that COWC data set meets our requirements. To remind a few, it is a *large* data set which contains thousands of diversified images of cars whose sizes that vary from 24 to 48 pixels match the values we are searching for ($\simeq 28$ *pixels*). For the reasons listed above, the COWC data set is choosen to be the *to go* algorithm for training our model.

Chapter 4

Training the model

In this chapter are described all the steps followed in order to train the network. It is possible to identify three distinct parts: **pre-processing**, **training** and **post-processing**.

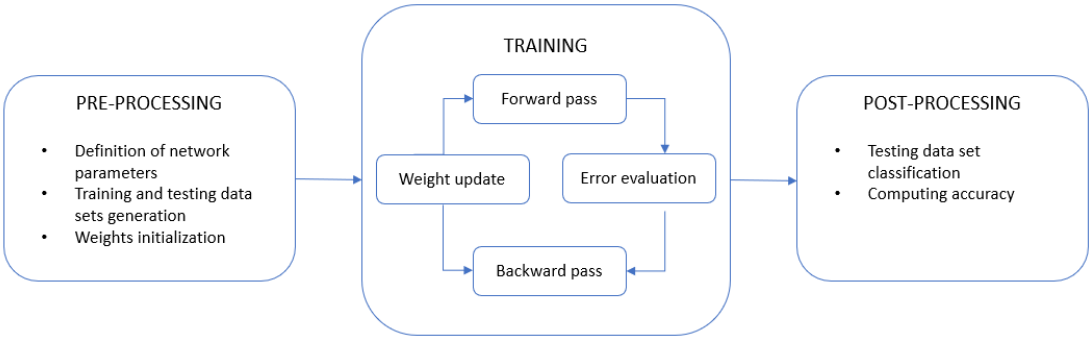

Figure 4.1: The training process

We previously illustrated that we are going to train the YOLOv3 model using Darknet. To achieve that, the following resources are needed:

- a powerful *Graphics Processing Unit* (**GPU**). GPU is a single chip processor used for extensive Graphical and Mathematical computations. In contrast to *Central Processing Unit* (CPU) which is suitable for serial instruction processing, GPU is suitable for parallel instruction processing. CPUs have few complicated cores which run processes sequentially with few threads at a time whereas, GPUs have large number of simple cores which allow parallel computing through thousands of threads computing at a time. This architecture reduces drastically the time required for training the model.
- **NVIDIA CUDA** and **cdDNN**. CUDA is a parallel computing platform and programming model developed by NVIDIA distribution for general computing on graphical processing units (GPUs). With **CUDA**, developers are able to speed up computing applications by harnessing the power of GPUs. The NVIDIA CUDA Deep Neural Network library (**cuDNN**) is a GPU-accelerated library for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers.
- **Open Source Computer Vision Library** which is a library of programming functions written in C++ that mainly aims at real-time computer vision
- **Darknet** which, as we have seen, is the open source neural network framework written in C and CUDA.

Google has recently provided the community with **Colab Virtual Machine (VM)**, which is a free cloud service based on Jupyter Notebooks that supports free 12 GB-RAM GPU and includes all of items we need to perform the training. Colab provides a run time fully configured for deep learning and free-of-charge access to a robust GPU. The main advantages of using this tool are:

- working directly from the files on the personal laptop;
- receiving the trained weights directly on the personal computer while the notebook is training;
- free-of-charge.

There are, however, some limitations. The first and main problem is that Colab run time is **volatile**. The Virtual Machine (VM) blows up after 12 hours and will disappear in the space. This means that the VM and all files are lost after 12 hours. After 12 hours the run time has to be reconfigured in order to start training again. This involves downloading all the tools, compiling libraries, uploading all the files and so on and so forth. Moreover, since we work on a remote VM there is not direct access to the VM file system. This means that we need to upload the files in order to be used and download the files during the training. To overcome these limitations, Google has included *Drive API* on the notebook that makes easy to map the user's Google Drive as a VM drive. This means that we have to synchronize one folder of the computer to Google Drive so that we can direct access to the Colab file system and test files instantly.

Figure 4.2: Colab Architecture

4.1 Pre-processing

To run Darkent we have to configure the runtime type on Colab to use GPU, get access to Google Drive, install cuDNN and clone and compile Darknet. After that, we need to get ready to train the model. To do so, we first need to prepare an image data set.
In order to better understand the impact of the training set on the training outcomes, the model will be trained and tested on two different data sets from COWC:

1. the first one includes the 2284 images taken from *Selwyn* data set;
2. the second one counts all the 25379 RGB colored imaged included in COWC (basically, it is an extension of the first data set);

In the post-processing section, we will further illustrated the motivation behind this choice along with the results.

4.1.1 Training and testing data sets generation

First of all, we need to specify YOLOv3 what images form the actual training set and what will serve as test set. For this purpose, a *test.txt* and *train.txt* files are generated through a Python script which splits the dataset into training and test images according to a certain percentage that we pass to the script – we set the percentage equal to 25%. We saw in the previous section that the object class in the file.txt of COWC data set is a number that goes from 1 to the number of classes. However, Darknet starts identifying the object class from 0 up to the number of $class - 1$. So before proceeding, we need to reduce by a factor of 1 all the first elements of the rows.

4.1.2 Definition of network parameters

YOLOv3 needs certain specific files to know how to train and what to train. For this reason, three files have to be created:

1. **obj.data**. It basically says on how many classes we are training the model on, what the train and the test set files are and what file contains the names for the category we want to detect. As it can be noticed from figure 4.3, the relative path points at the Google Drive folder where all the files and images have to be first uploaded. In section

```
classes= 4
train = /content/gdrive/My\ Drive/darknet/train.txt
valid = /content/gdrive/My\ Drive/darknet/test.txt
names = /content/gdrive/My\ Drive/darknet/obj.names
backup = /content/gdrive/My\ Drive/darknet/backup/
```

Figure 4.3: The obj.data file

2. **obj.names**. It contains a column with the name of the classes (*Sedan*, *Pickup*, *Others*, *Unknown*) we are training the model on.

3. **yolo.cfg**. This file contains the definition of all the layers the neural net is made of (which are 106) and some important network parameters. Among the most meaningful are:

   - **batch** indicates how many images and corresponding labels are used in the forward propagation to compute a gradient and update the weights via back propagation. If batch = 64, the model is loading 64 images for one single iteration.
   - **subdivisions** indicate in how many *blocks* the batch is subdivided. The images of a block are ran in parallel on the GPU. If batch = 64 and subdivisions = 8, the batch is split into 8 *mini-batches* which means that 64/8 images get sent to the GPU. This will be repeated 8 times until the batch is completed and a new iteration will start with 64 new images. When batching the intend is not only to speed up the training process but also to generalize the training more. If the GPU is powerful with loads of RAM, this number can be decreased, or batch could be increased.
   - **width and height**. These configuration parameters specify the input image size. The input training images are first resized according to width x height before training.
   - **channel** indicates the number of channel of the input images. If channel is equal 3, 3-channel RGB input images will be processed.
   - **learning rate** is a parameter that determines how much an updating step influences the current value of the weights. Typically this is a number between 0.01 and 0.0001. At the beginning of the training process, we are starting with zero information and so the learning rate needs to be high. But as the neural network sees a lot of data, the weights need to change less aggressively. In other words, the learning rate needs to be decreased over time. In the configuration file, this decrease in learning rate is accomplished by first specifying that our learning rate decreasing *policy* is *steps*.
   - **momentum**. In previous sections, we mentioned how the weights of a neural network are updated based on a small batch of images and not the entire data set. Because of this reason, the weight updates fluctuate quite a bit. That is why a parameter *momentum* is used to penalize large weight changes between iterations.

- **decay** is an additional term in the weight update rule that causes the weights to exponentially decay to zero. Practically, after each update the weights are multiplied by a factor slightly less than 1. This prevents the weights from growing too large. A typical neural network has millions of weights and therefore they can easily overfit any training data. *Overfitting* means it will do very well on training data and poorly on test data. It is almost like the neural network has memorized the answer to all images in the training set, but really not learned the underlying concept. One of the ways to mitigate this problem is to penalize large value for weights.
- **filters** indicates how many convolutional kernels there are in a layer.
- **activation** indicates the type of activation function which is applied to the weighted sum.

Figure 4.4 (a) shows the first lines of our *yolo.cfg* file, where the neural network parameters are defined for training our model on the first data set (2284 images). We started the training with batch = 64 and subdivision = 1 and we got an out of memory error meaning that the GPU has not enough memory to load 64 images at a time. To overcome this limit, the number of subdivisions has been increased by multiple of 2 (e.g 2, 4, 8) till the training proceeded successfully for subdivisions = 8. On the other hand figure 4.4 (b) shows the first lines of the *yolo.cfg* file for the training of our model on the entire data set (25379 images). We can see that after some tryouts, the only combinations of batch and subdivisions that did not cause an out of memory error was batch = 32 and subdivisions = 32. During testing, both batch and subdivisions are usually set to 1 regardless the data set.
As regards the input image size, we use the default values of 416×416. The results might improve if we increase it to 608×608, but it would take too longer to train.
The learning rate will start from 0.001 and remain constant for 3800 iterations, and then it will multiply by scales to get the new learning rate. In the previous paragraph, we mentioned that the learning rate needs to be high in the beginning and low later on. While that statement is largely true, it has been empirically found that the training speed tends to increase if we have a lower learning rate for a short period of time at the very beginning. This is controlled by the **burn_in** parameter.
The **angle** parameter in the configuration file allows us to randomly rotate the given image by its value. Similarly, if we transform the colors of the entire picture using **saturation**, **exposure**, and **hue**, it will still be the picture of a car. We used the default values for training.

```
# Testing
# batch=1
# subdivisions=1
# Training
batch=64
subdivisions=8
width=416
height=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.001
burn_in=1000
max_batches = 500200
policy=steps
steps=400000,450000
scales=.1,.1
```

(a)

```
# Testing
# batch=1
# subdivisions=1
# Training
batch=32
subdivisions=32
width=416
height=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.001
burn_in=1000
max_batches = 500200
policy=steps
steps=400000,450000
scales=.1,.1
```

(b)

Figure 4.4: The yolo.cfg files for the two training processes

4.1.3 Weights initialization

When training an object detector, it is recommend to leverage existing models trained on very large data sets even though the large data sets may not contain the object we try to detect. This process is called transfer learning. For this reasons, instead of learning from scratch we use a pre-trained model which contains convolutional weights trained on **ImageNet** which is a large visual database designed for visual object recognition software research. By sing these weights as our starting weights, our network will learn faster.

4.2 Training

Once all the parameters are set, the training process can start. As the training goes one, Darknet saves in a backup folder the network weights after every 100 iterations till the first 1000 and then saves only after every 1000 iterations. The update weights will be later processed to estimate the accuracy of the training. The first part of the terminal output looks like in figure 4.5 where the first layers of the framework are shown along with the number of filters, input and output dimensions.

```
yolov3
layer     filters    size              input                output
   0 conv     32  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  32 0.299 BF
   1 conv     64  3 x 3 / 2   416 x 416 x  32   ->   208 x 208 x  64 1.595 BF
   2 conv     32  1 x 1 / 1   208 x 208 x  64   ->   208 x 208 x  32 0.177 BF
   3 conv     64  3 x 3 / 1   208 x 208 x  32   ->   208 x 208 x  64 1.595 BF
   4 Shortcut Layer: 1
   5 conv    128  3 x 3 / 2   208 x 208 x  64   ->   104 x 104 x 128 1.595 BF
   6 conv     64  1 x 1 / 1   104 x 104 x 128   ->   104 x 104 x  64 0.177 BF
   7 conv    128  3 x 3 / 1   104 x 104 x  64   ->   104 x 104 x 128 1.595 BF
```

Figure 4.5: Terminal output while training

The output below (4.5) is also displays by Darknet detector. In this way, we can monitor the loss while the training is proceeding.

```
 17142: 0.747216, 1.126627 avg loss, 0.001000 rate, 10.445256 seconds, 548544 images
Loaded: 0.000095 seconds

 17143: 1.200003, 1.133964 avg loss, 0.001000 rate, 10.451601 seconds, 548576 images
Loaded: 0.000057 seconds
```

Figure 4.6: Batch output while training

Analyzing the first line of code:

- < 17142 > indicates the current training **iteration**. Iteration is one time processing for forward and backward propagation of a batch of images. If batch = 32, then 32 images are processed in one iteration.
- < 0.747216 > is the **total loss** which is a number indicating how bad the model's prediction was the current batch
- < 1.126627 avg > is the average loss error till the current batch.
- < 0.001000 $rate$ > represents the current learning rate, as defined in the $.cfg$ file.
- < 10.445256 $seconds$ > represents the total time spent to process the current batch.

- < 548544 > is the total amount of images that passed through the neural network so far. It is equal to the number of iterations times the images of the batch ($17142 \times 32 = 548544$). As we can see considering the second line of code, the number of images has increased by the batch value.

As the training goes on, a file called *train.log* in the data set directory contains the loss registered in each batch. In this way it is possible to plot the loss against the batch number. By plotting these two quantities, we can have a measure of when the training should stop and it is common practice to stop the process after the loss has reached below some threshold. The final average loss can be from 0.05 (for a small model and easy data set) to 3.0 (for a big model and a difficult data set). Usually 2000 iterations are sufficient for each class (so in our case 8000), but not less than 4000 iterations in total. If we consider the first training data set (2284 images), the trend of the average loss as the learning goes on is the one shown in the figure below. The high peak when the batch number is relative low indicates that the model is further to be accurate on our custom data set. This result is expected since we initialized the model with the weights taken from another training process. But as the number of batches increases and the model learns how to correctly detect our customized objects, the average loss decreases until it reaches approximately 0.2549. After this value the average loss is holding stead and we can stop training.

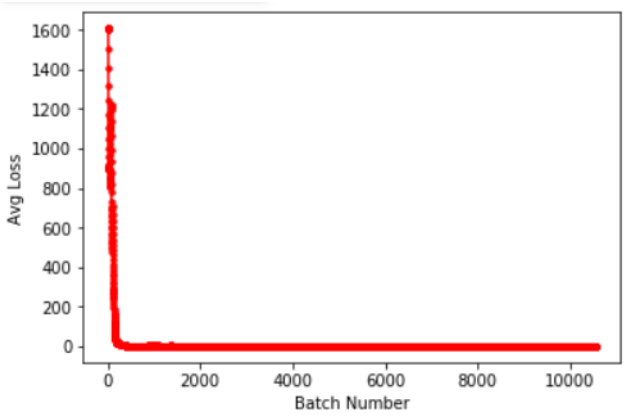

Figure 4.7: Average loss against batch number for the first training

4.3 Post-processing

Once training is stopped, we need to evaluate the accuracy of the model. Considering the testing data set, the network is used to classify each of the images within the data set. The true category to which each of the images belongs is compared to the output category provided by the network. In pattern recognition, the accuracy of a model is computed in terms of **precision** and **recall**. According to the following equation, the two quantities are defined as:

$$precision = \frac{tp}{tp + fp} \tag{4.1}$$

$$recall = \frac{tp}{tp + fn} \tag{4.2}$$

where tp stands for true positive, while fp and fn indicate false positives and false negatives respectively. It is easier to visualize the definition of precision and recall by looking at figure 4.8.

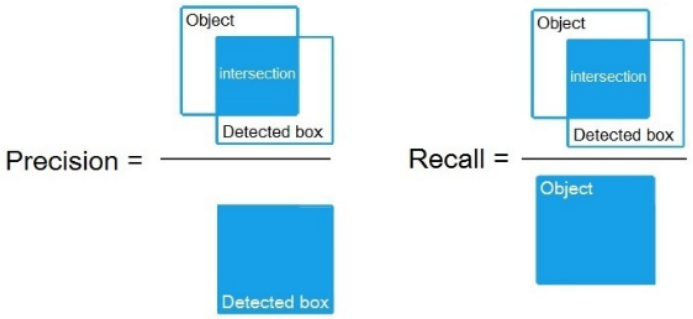

Figure 4.8: Precision and Recall

To get the accuracy, we take some of the weights from the backup folder and we compare them in order to choose the ones that produce the best results. Not always the last weights turn out to be the most accurate. Up until a certain number of iterations, new iterations improve the model. After that point, however, the model's ability to generalize can weaken as it begins to overfit the training data. A model is overfitted when "it corresponds too closely or exactly to a particular set of data and may therefore failed to fit additional data". This may happen when a neural network is trained to classify in an extremely precise way all the training data set, but fails to classify an other data set.

4.3.1 Testing cases

First, we trained the model on the smallest data set which includes the images taken from Selwyn. After that, we tested the model on actually two testing sets:

1. the first is formed by the 25 % of the images of the Selwyn data set that we previously split into training and testing according to this percentage.

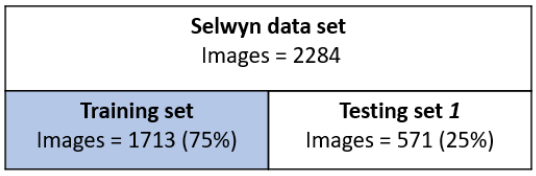

Figure 4.9: Shuffling Selwyn data set into training and testing set

2. the second includes all the images of COWC except the 75 % of Selwyn that have been employed for training the model.

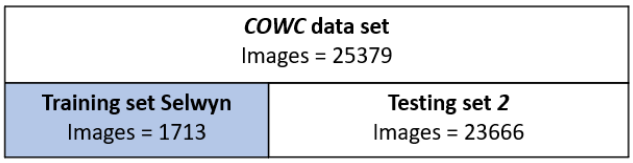

Figure 4.10: Testing set 2 for Selwyn

Moreover, we set a **confidence threshold** that measures how confident is the network about the prediction of a bounding box – in other terms, it measure the probability of the bounding box to predict precisely for the object class.
Along with choosing different weight for each test case, we make this threshold varying between 0.5 and 0.7 and compare the different results.

1. *Case 1*: **Testing set 1 – Confidence Threshold 0.7**

Iterations	Precision	Recall
1000	0	0
3000	94,18%	69,12%
8000	96,38%	79,40%
9000	98,31%	79,40%
10000	96,56 %	78,97

Table 4.1: Precision and Recall for case 1

2. *Case 2*: **Testing set 1 – Confidence Threshold 0.5**

Iterations	Precision	Recall
1000	0	0
3000	92,54%	72,93%
8000	93,78%	80,94%
9000	96,36%	81,96%
10000	94,08 %	79,82

Table 4.2: Precision and Recall for case 2

By looking at tables above, we can state that precision is always higher then recall in all testing cases. In other words, this means that when the model predicts a bounding box the probability that it corresponds to the ground truth is higher (*precision*) while the probability that the model predicts all bounding boxes is lower – it occurs that the model does not predict some bounding boxes (*recall*). From case 1 to case 2 where the confidence threshold is decreased, precision is decreasing while recall is increasing. As we can expected, also bounding boxes with lower confidence contribute to accuracy (recall increases) but it may happen that not all of them predict cars (precision decreases).

From table 4.1 and 4.2 Weights corresponding to 9000 iterations are the ones that produce the best results. The model predicts properly for Selwyn that, however, is a small sample of huge collection. Even if the features are similar among the entire data set, this model may not produce satisfying results if tested on all the images of COWC. For instance, changes in light conditions that may not be even visible to

human eyes could lead the model to fail. For this reason, we tested the model on a bigger testing set that, as we previously showed, includes all the images within COWC excepted the ones we trained the model on.

3. *Case 3*: **Testing set 2 – Confidence Threshold 0.7**

Iterations	Precision	Recall
9000	96,81%	39%

Table 4.3: Precision and Recall for testing case 3

As expecting, the precision is still high while the accuracy in terms of recall is drastically decreased. That means that approximately the 60 % of the times, the model fails to detect cars. This result is, of course, not acceptable because a failure in the detection will cause a failure of the entire tracking. Here comes the need to repeat the training on the entire COWC data set in order to increase the "knowledge" of our model. Again, the data set has been split into training (75 %) and testing (25 %) set as figure 4.11 shows.

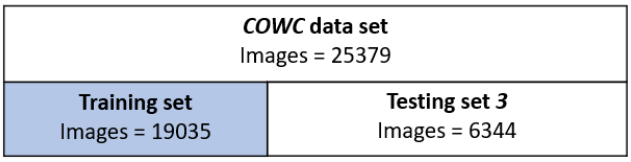

Figure 4.11: Shuffling COWC data set

Since we dealt with many more images, the second training required three days and about 19000 iterations to converge. Once the training stopped, we tested the model on testing set 3 and got the following results.

4. *Case 4*: **Testing set 3 – Confidence Threshold 0.7**

Iterations	Precision	Recall
4000	3,50%	0,91%
8000	95,38%	49,80%
10000	96,06%	55,85%
13000	95,50%	58,46%
17000	97,57%	62,35%
19000	96,21%	63,13%

Table 4.4: Precision and Recall for testing case 4

5. *Case 5*: **Testing set 3 – Confidence Threshold 0.5**

Iterations	Precision	Recall
4000	2,88%	1,37%
8000	94,66%	50,06%
10000	95,85%	56,24%
13000	95,31%	58,99%
17000	96,47%	63,10%
19000	95,54%	64,29%

Table 4.5: Precision and Recall for testing case 5

As we can see, the precision is still around 95% while the recall increased by the 30%. Even if the accuracy is not optimal, this results represent a good staring point for our projects. After this statistical computations, we wanted to test our model by manually. We fed it with some aerial images – some downloaded from the web and others of real past flight missions. Here are some results. In both cases, the model works fine. In the first image, both cars are detected and the confidence is high (93%). The second image is more chaotic but nevertheless the outcome is pretty accurate (just the upper line of car is not seen by the model, probably because it is cropped by the image itself).

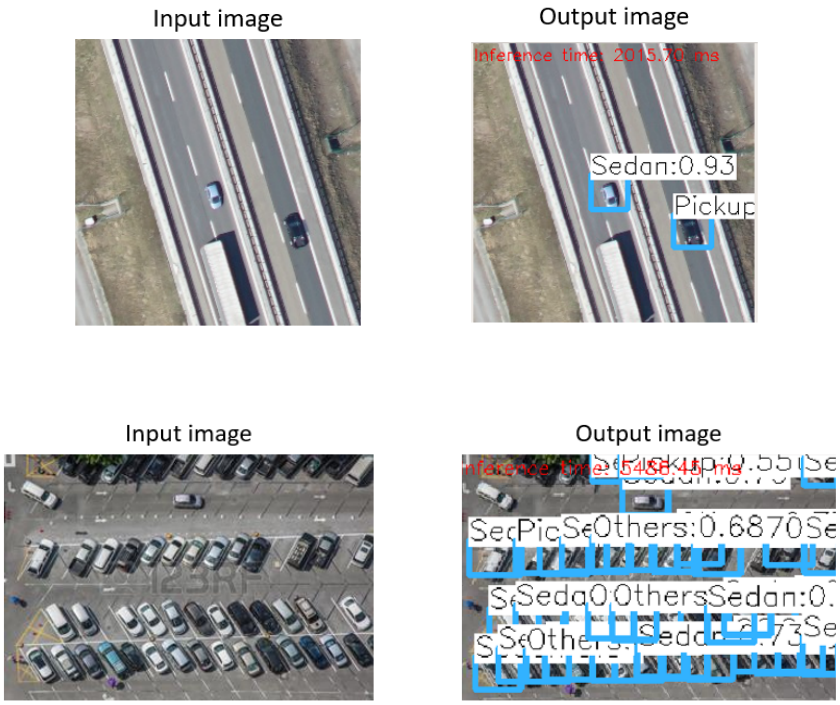

Figure 4.12: Testing the model on real aerial images

Chapter 5

Implementation

In this chapter, it will be illustrated the implementation of the system described so far in terms of hardware and software.

5.1 Hardware

Figure 5.1 shows a sketch that includes the components used for a first implementation of the tracking system. It is important to highlight that this is a preliminary set-up and some components, such as the sensor, will be different in the final product. However, this arrangement allows us to eventually validate all the logic described by the software. The coming sections will provide a description of the components and how they communicate between them.

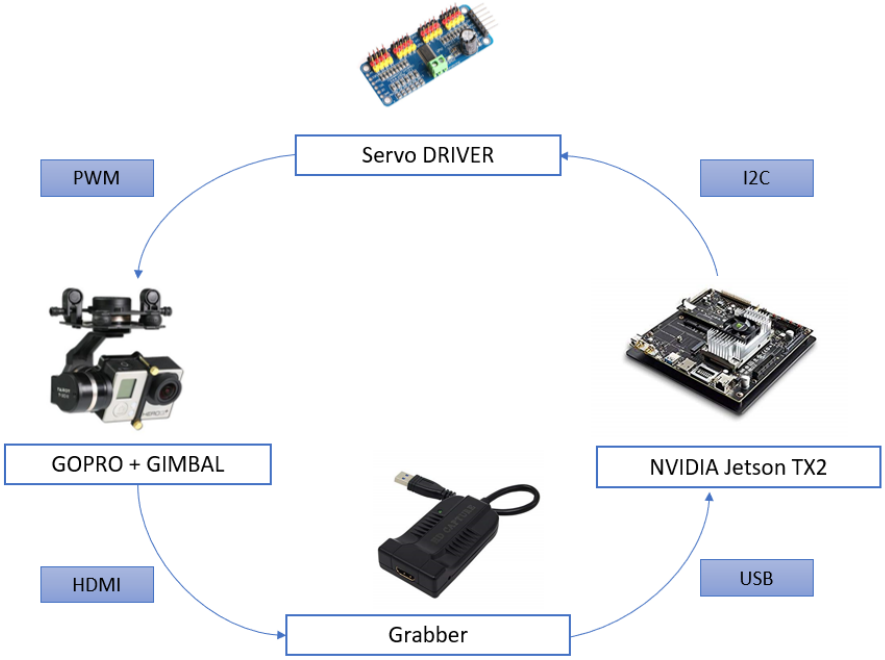

Figure 5.1: Hardware configuration

5.1.1 NVIDIA Jetson TX2

NVIDIA Jetson TX2 is the fastest and most power-efficient embedded AI computing boards from NVIDIA. The board is provided with a 256-core NVIDIA Pascal GPU architecture with 256 NVIDIA CUDA cores. This means that we can perform definitely more parallel operations during the detection step. As explained before, AI

highly benefits from parallel GPU computations. Practically, for deep learning real time systems GPUs are compulsory. Our project has a strong requirement in terms of throughput which means that we need to process a large number of images in the unit of time. We want to detect as fast as possible an object in order to increase the precision in the tracking system. If we had a huge delay in the detection, it would mean that in then meanwhile the object has moved and we may loose the track. For all these reasons, we decide to exploit a more specific hardware support (GPU) in order to enhance our overall system performances.

The board features a variety of standard hardware interfaces (HDMI, USB, Ethernet) (figure 5.2) that make it easy to integrate it into a wide range of environments.

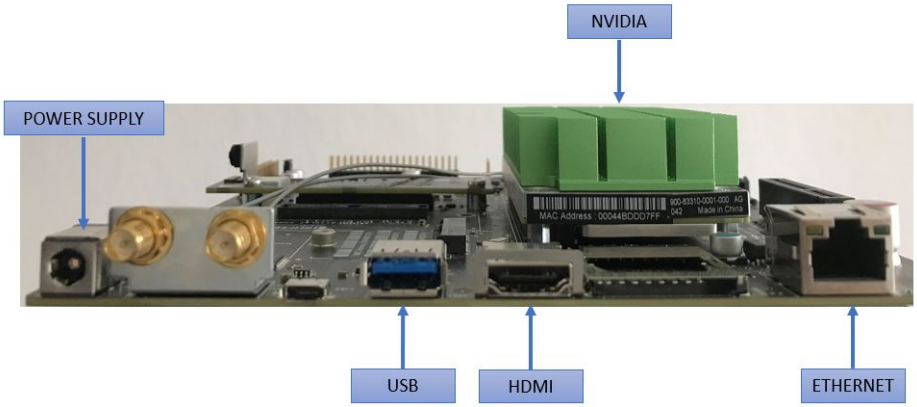

Figure 5.2: NVIDIA interfaces – frontal view

The board is also provided with *general-purpose input/output* (**GPIO**) pins. Precisely, there are 40 pins in the GPIO Header and the numeration starts where the arrow is pointing at in figure 5.3.
GPIO do not have a specific and unique task. Depending on the contest, GPIO function can be customized. Some of its connections can implement **I2C** (*Inter-Integrated Circuit*) protocol, which is a master-slave serial bus. In fact, transmitting and receiving data between two or more devices (in our project between the NVIDIA board and the servos) requires a communication path called *bus*.

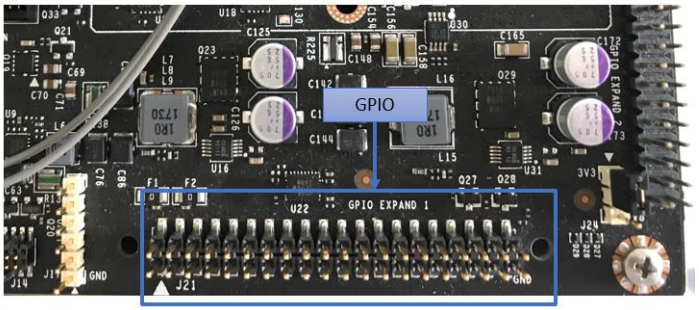

Figure 5.3: GPIO on NVIDIA board

I2C basically uses two bidirectional wires, one called *Serial Data Line* (**SDA**) for transferring data and one called *Serial Clock Line* (**SCL**), which is the clock that sets the start and the stop of the data transfer. Most of the times, also a wire with *Voltage at the Common Collector* **VCC** and *Ground* (**GND**) is required. Each node of the bus has a role: it can be either master or slave. The former is the one that creates the clock and starts the communication with the slave, while the latter is the one the clock is addressed to.

On NVIDIA Jetson TX2 it is possible to access either I2C bus 0 or I2C bus one. The wiring combination to access I2C bus 1 is the following:

- GND is on pin 6;
- VCC is on pin 2;
- SDA is on pin 3;
- SCL in on pin 5;

Name	Pin	Pin	Name
3.3 VDC *Power*	1	2	**5.0 VDC** *Power*
SDA1 *General I2C #0 Data 3.3.V, I2C Bus 0*	3	4	**5.0 VDC** *Power*
SCL1 *General I2C #0 Data 3.3.V, I2C Bus 0*	5	6	**GND**
GPIO_GCLK *AUDIO_MCLK Audio Master Clock (1.8/3.3.V)*	7	8	**TXD0** *UART #0 Transmit*

Figure 5.4: Pin access to I2C bus 1

5.1.2 Servo DRIVER

As we saw in section 2.2, in order to track a target the gimbal has to rotate on its pitch and yaw axis. In general, it can be challenging to control an analog device when the signal to drive it is generated by a digital component. One method to convert digital into analog is the *digital to analog converter* that however adds complexity to a project. An alternative and easier method to drive the servo is by *Pulse-width modulation* (**PWM**) that as the name itself suggests, consists in pulsing the digital signal high and low at a fast rate. The resulting signal reproduces an effective average voltage. To enable the communication between the NVIDIA board which communicates through low powered signal (max 5 volts) and the servo, which is driven by pulse-width modulation, we need an intermediate module. This is a 16-Channel **PWM Servo motor driver**. This servo needs two I2C pins (that become 4 if we consider VCC and GND) in order to drive up to 16 up PWM channels. Figure 5.7 shows the module interfaces. According to the wiring combination for I2C bus 1 on NVIDIA, the black wire on the left is plugged into pin 6, the red wire into pin 2 and so on.

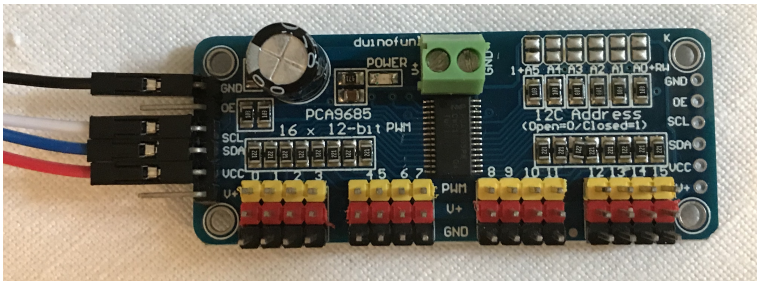

Figure 5.5: The 16-Channel **PWM Servo motor driver** interfaces

5.1.3 Gimbal and GoPRO Camera

The NVIDIA board is provided with an embedded camera that however cannot be mounted on a servo. For this reason, we had to choose for another solution. To exploit the resources made available by the company, we opted for employing the Tarot T4-3D three-axis-stabilized gimbal suitable for GOPRO. It has a **FPV Mode** which means that the gimbal is synchronized with the movement of the aerial platform (aircraft or UAV) it is mounted one. For this feature, it provides a first-person perspective flying experience (which is actually what our tracking system is supposed to do).
As regards the most relevant technical specs, the construction of the gimbal is made of carbon fibers, aluminum and fiber reinforced plastic. There are damping balls which are are soft enough to reduce vibrations. As regards the angles, the **yaw** control is limited to approximately 125°, maximum **roll** angles are ± 45° while maximum **pitch** angles are +25° and -114°. There is a gimbal controller where signal wires for video output, mode switching and tilt control are connected together. The video link could be used to send the video signal to an hypothetical ground station for live video monitoring. The controller is connected to the gimbal by a flexible multi-cores cable.

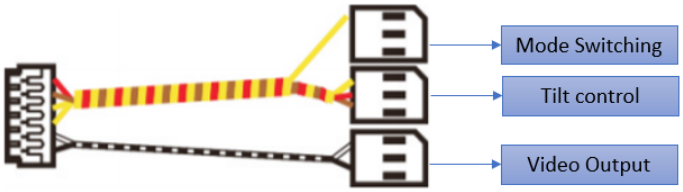

Figure 5.6: Tarot T4-3D wires connections

5.1.4 HDMI to USB Frame Grabber

The grabber is a device that captures video from an HDMI Source (in our project the GoPRO camera) and transfer its content to a computer (in our project the NVIDIA board) via USB.

Figure 5.7: Tarot T4-3D wires connections

5.2 Software

This section includes the implementation of the modules that will be loaded onto the NVIDIA board in order to perform all the steps of our tracking system. These are:

1. *video streaming.*

   This module opens a communication with the camera and displays what the sensor is pointing at. The requirement of this module is to process as fast as possible all the frames that the camera is addressing to it. Usually the speed at which a camera can capture frames varies from 60 to 120 frames per second (fps).

2. *object detection*

   This module performs car detection on frames extracted from the streaming. Therefore, the requirement is to extract frames (on request) on which it will apply the model we trained. Ideally, it can process 45 frames per second.

3. *angles computation*

   This module computes the mismatch angles in order to have the camera pointing at the detected object.

4. *movement of the motor*

   This module moves the motor accordingly to the angles computation outcomes. The requirement for this module is to keep up with the frame extraction and object detection. However, the rate at which this process progresses is limited by the speed of motor itself. For this reason, this module is categorized as **I/O intensive** because most of the time the CPU on which it is running waits for the motor to stop in order to ask for another frame.

From the description above, it is evident that the requirement of the motor movement is not satisfied if one single process coordinated all the modules. In fact, the streaming video would be paused while waiting for the motor to end its stroke. This latency is of course not acceptable because we would not know what the camera is recording and we may loose the track. To overcome this limits, we implemented a **client-server architecture**. This model distributes workloads between the providers of a resource, called *servers* and those who require it, called *clients*. Servers and clients communicate through messages. In particular, the client sends a request to the server who returns a response after processing the request. This way of communicating is defined **inter-process communication**.

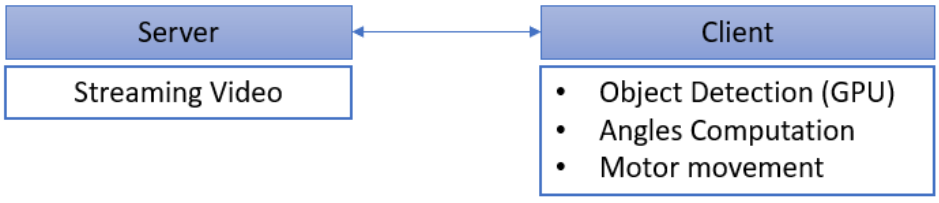

Figure 5.8: Software architecture

According to this architecture, the diagram of the processes can be visualized in figure 5.8. As we can see, the module that menages video streaming can be identified as the *server*. The streaming module serves, on request, the unit that performs object detection by sending the current frame of the streaming feed. If in this frame a car is detected, the angle computation module computes the mismatch angles that are eventually communicate to the functional unit that moves the motor. After the motor has moved, the client will require for another frame, the detection modules will search for a car and, if present, angles computation and motor movement modules are

run. If none is found, another frame will be asked to the streaming until one car is detected and so forth. It is important to highlight that while the client is requesting and processing frames, the streaming module is not paused and keeps displays what the camera is shooting.

Chapter 6

Validation

Due to some setbacks (some hardware was missing), it was possible to validate the streaming and detection modules but not the movement of the camera.

The set up configuration was the following:

- the sensor used was the camera embedded on the NVIDIA board. It is fixed and cannot rotate but for the detection purposes this works fine;
- the output video of the board was connected to a monitor is order to visualize the streaming and the detection output;
- the targets used for the detection were a red and gray car miniatures;

The set up was hold by a wooden structure in order to fix all the devices. The validation was performed outdoor, in the daytime so that the lighting conditions were as similar as possible to the ones of the COWC data sets.
First, the car miniature has been placed on the street while the camera was positioned upward. Car size and distance from the camera were chosen so that the car appeared approximately 30 pixels. Figure 6.1 shows that the detection algorithm returns a bounding box around our red car miniature with the confidence of 75%.

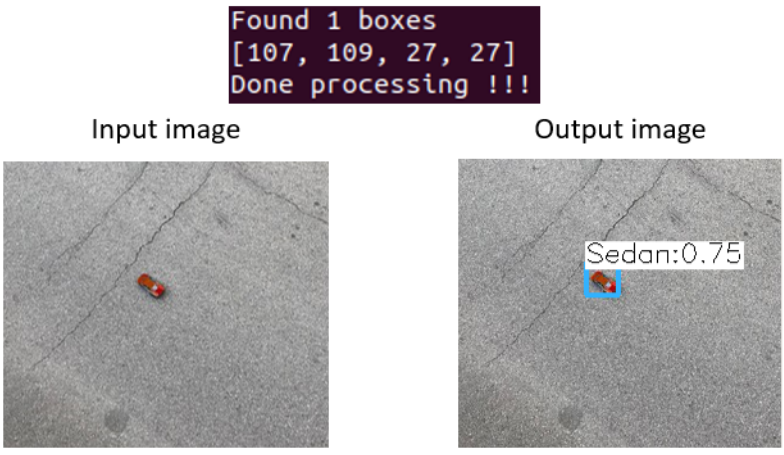

Figure 6.1: First validation case

After that, we repeated the validation on the gray car miniature. As we can see in figure 6.2, our model still successes in detecting the car (with the confidence of the 85%) even if in this scenario the contrast between target and *context* is pretty low. This satisfying result is expected since the training sets from which our model learnt is thought to be difficult in terms of occlusion and low context.

Figure 6.2: Second validation case

6.1 Further Developments

Further developments of the prototype proposed in this thesis regard the validation of the angles computation module and movement of the camera both in a stationary and dynamic scenario where target and camera are in relative motion. We may expected that the algorithm and the implementation are not perfect so we may need to revisit the approach. Moreover, the tracking algorithm cannot handle occlusions. For this reason an inference probabilistic approach is likely to be taken into consideration to overcome this limit. Once these results are successfully achieved, the accuracy of the detection model should be enhance by training the model on a more meaningful data set. This implies collecting ourselves the data from real flight missions. Therefore, the model will be trained on more accurate data according to our operative environment. Moreover, the classes of target that the model is able to recognized can be enlarged (not only cars). This means that the training has to be repeated on a larger data set that includes many labeled images of the unseen object classes.

Bibliography

[1] David Forsyth, Jean Ponce. *Computer Vision: A Modern Approach*

[2] `https://en.wikipedia.org/wiki/Object_detection`

[3] `https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)`

[4] `https://www.analyticsvidhya.com/blog/2018/12/practical-guide-object-detection-yolo-framewor-python/`

[5] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection.*

[6] Mark Everingham, S. M. Ali Eslami, Luc Van Gool, Christopher K. I. Williams, John Winn, Andrew Zisserman. *The pascal visual object classes challenge: A retrospective.* International Journal of Computer Vision.

[7] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrenc Zitnick, Piotr Dollar *Microsoft COCO: Common Objects in Context*

[8] Computational Engineering Division, Lawrence Livermore National Laboratory *A Large Contextual Dataset for Classification, Detection and Counting of Cars with Deep Learning.*

[9] Franklin Tanner, Brian Colder, Craig Pullen, David Heagy, Michael Eppolito, Veronica Carlan, Carsten Oertel, Phil Sallee. *Overhead imagery research data set: an annotated data library and tools to aid in the development of computer vision algorithms.*

[10] Sebastien Razakarivony and Frederic Jurie. *Vehicle Detection in Aerial Imagery (VEDAI) : a benchmark*