Università
Ca'Foscari
Venezia

Major to
Computer Science

Master Degree Thesis

Ca' Foscari
Dorsoduro 3246
30123 Venezia

drAIver: A Simple Self-Driving Robot

Advisor
Chiar.mo Marcello Pelillo

Co-advirsor
Dr. Sebastiano Vascon

Graduand
Marco Signoretto
Registration number 850484

Academic year
2017 / 2018

Abstract

In the last few years, autonomous driving has become a major topic in both car industry and academia. Many challenging problems remain to be solved, however, in order for autonomous cars to be on the market. In this thesis, I will describe the structure of a simple robotic system, called *drAlver*, which is able to navigate autonomously using computer vision algorithms. To reach this goal the problem has been decomposed into tasks. The first task is road line detection, two kinds of detector have been proposed. A *basic line detector* and an *advanced line detector*. The latter method shows more effective performances overcoming some limitations of the first one. The second one is the detection of cars, pedestrians, cyclists and traffic signs which is solved with a CNN, in particular using the YOLOv3 architecture. The above tasks are not solved on the robot's board due to its limited computational power. Instead, the robot is paired with a computer via a wireless connection. The thesis work is also related to the engineering and the development of the hardware and the software as well as the communication structure between computer and robot. The hardware is based on *Raspberry Pi Model B*, which is able to control motors. To sense the surrounding environment, the robot captures images using a webcam positioned on its top. The communication between different modules is based on queues resulting in an asynchronous and parallelized system.

Contents

Chapter 1

Introduction

The project's goal consists in realizing a simple autonomous driving robot. All the code is Open Source and it is available on GitHub [34].

This robot is not a complete autonomous vehicle, but it executes some of the main tasks of an autonomous driving car. To performs these tasks the robot needs to be connected to a Desktop or laptop PC with NVIDIA Graphics card and CUDA capabilities.

The thesis focusses on three tasks: the lane detection, the object recognition of road's traffic signs and road's obstacles, such as cars, pedestrians and bicycles. The final task that has to be solved is related to the motion in the simulated environment, given the informations obtained from the previous tasks.

All the work presented in this thesis focuses on real-time performances, using cheap and not powerful hardware, for this reason the first task has been solved using classical computer vision approaches and not deep learning ones, that although they are more effective in terms of reliability, they demand lots of computational capabilities.

The objected detection tasks are based on YOLOv3 architecture. To solve this task two different networks have been used trained respectively on LISA [25] and KITTI [14] datasets.

For lane detection two approaches have been developed, where the second one comes from the limitations of the first. Both consist on a pipeline of consecutive filtering operations and are based on classical computer vision techniques as linear filtering, bird view and adaptive threshold.

The motion task instead is based on real-time Object Oriented World Model proposed by [12].

In literature there is not a self-driving robot prototype which is able to solve these basic self-driving tasks using only monocular camera and which is able to run

in real-time on both GPU and CPU computers with high reliability.

In the project different approaches have been used combining deep learning approaches with classical computer vision ones. All the project is written in **Python3** with C++ core used by libraries like *Numpy*,*Darknet*,*Tensorflow* and *opencv* to guarantee good performances.

1.1 Thesis structure

The first chapter is related to the first task which is the lane detection, for it two solutions have been proposed: the first one is not applicable to real usage, but it is useful because thanks to that the second one more effective solution has been proposed. Both techniques are based on classical Computer Vision approaches. In the chapter it will be explained how the algorithms work and their limitations.

In the second chapter will be presented the object detection task, this will be performed using deep learning techniques. Since the objects that we want to recognize are present in different datasets for simplicity two networks have been used. To solve this task several deep learning architectures have been considered and after some competitive analysis the tiny YOLO v3 architecture has been chosen, the other strong candidate was the SSD [24] based on MobileNet [19]. We will discuss the training procedures and how the final detector works. The two network have been trained respectively on LISA [25] dataset for the traffic signs recognition and on KITTI [14] for cars, pedestrians and cyclists detection.

On the third chapter the focus moves to the motion and decision making. There will be described the techniques used to store pieces of information and how retrieve them in real-time, as well as representing robot state related to the actions that it must performs.

Moving at an higher level on chapter 5 it is presented the whole system architecture. Starting from communication protocols used to interface computer and robot until the internal architecture of the software which runs on the computer and the one running on robot. The internal software architecture will be described in terms of modules and how they interact with each other to solve all the tasks.

In the next chapter, which is the chapter 6, there is the description of all the hardware used and where you can find the references for the setup instructions for both, computer and robot with related library discussions.

On the last chapter (chapter 7) before conclusion performances' analysis of the algorithms used on the above chapters have been reported. In particular the manually generated dataset used to evaluate in a quantitative way the performances

of the second lane detector algorithm will be presented and the some common metrics like mAP will be evaluated for the object detection algorithm for both the networks.

Chapter 2

Lane detection

This chapter will explain the approaches that have been used to identify road lanes.

Today the autonomous driving tasks are well studied and lots of research articles are available. All the recent approaches use deep learning methods in particular these approaches are focused on road segmentation which is solved with image segmentation, these methods use CNN to predict the masks of the road as in [38].

In [22] a method has been provided to handle lane and road marking detection and recognition that is guided by a vanishing point under adverse weather conditions.

This approach is very effective and the source code are available on GitHub at `https://github.com/SeokjuLee/VPGNet`. Authors say that the network is able to run on real-time with 20fps on a computer equipped with NVIDIA Titan X.

The above articles are very interesting from the theoretical point of view, but both use Deep Neural Networks, since in this project we have a limited amount of resources like RAM, CPU and GPU which is very far away from the performance that we can obtain with a NVIDIA Titan X; these deep approaches are unfeasible for us.

In this project it is necessary to make some compromise, this one is related to resource usage, and deep learning approaches need lots of resources in terms of memory and computational power in particular to run in real-time.

Since in this project we need also to recognize cars, pedestrian, bicycles and traffic signs, the use of deep learning approaches has been used on such tasks as discussed in chapter 3, so for the lane detection a classical computer vision approach has been chosen.

Another fact that has affected our decision is related to the fact that training deep network is a very expensive task in terms of time as demonstrated in chapter 3.

For the lane detection task, in this project, two approaches have been implemented and as final result the second one has been used.

The algorithms have been tested on road images from *Kitty Dataset* [14], in this dataset there are lots of images with difficult lighting conditions and shadows.

2.1 Basic lane detector

Here we will discuss the first approach that has been used to develop the lane detector, this approach has lots of limitations and for this reason on section 2.2 will be presented a more effective and reliable approach.

The explanation of this approach will be fast, since this approach is not a candidate for real usage, but it is used only in order to analyse its limitations which are used to introduce the second one.

This approach is based on the idea that lines, also curved, can be approximated to strict ones, so the idea is to be able to find two strict lines that approximate the road lanes and use them to follow the road.

The proposed algorithm pipeline is the following:

- Pre-Filtering
  - Convert frame into grey scale
  - Adaptive Thresholding
- Line searching
  - Hough lines algorithm
  - Lines angle filtering
  - DBSCAN Clustering

2.1.1 Pre filtering

The coloured images are captured, converted to gray scale and then thresholded. As first attempt an Otzu thresholding was applied, but this kind of thresholding was not effective for the problem. Roads are often subject to difficult lighting conditions like shadows, darker roads than the background and sun reflection. Under these conditions Otzu performs very bad since it works on the histogram of the whole image so it tends to separate areas without sun illumination from areas with sun illumination. To overcome such problem an adaptive thresholding has been used; this kind of thresholding works on local information mitigating the problem above.

An example of the results of the two approaches are shown in figures 2.1 and 2.2 where it's evident that adaptive approach works well in this kind of lighting conditions.

Figure 2.1: Example of road image thresholded with Otzu algorithm

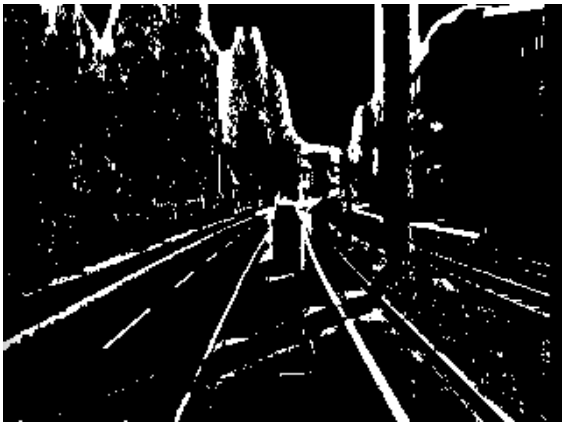


Figure 2.2: The same image thresholded with adaptive mean algorithm

2.1.2 Line searching

At this point, given a binary thresholded image, the idea is to use the *Hough transform* algorithm ([35] section 4.3.2) to search lines on the image. The vote threshold that has been used is 200, after that lines have been filtered respect to the line angle, so only lines with a θ between 0.78 and 2.35 will be kept.

A lot of lines have been extracted at this stage, so in order to cluster together closed lines a clustering algorithm needs to be applied.

The desired clustering must be performed on two dimensions, since the *Hough transform* works on $\rho - \theta$ plane the clustering will be applied on such dimensions, the number of clusters is not known, so the candidate algorithm is an algorithm that is able to propose the number of clusters. Another requirement is that it is able to handle non globular clusters because, as we can see from figure 2.3, the clusters are elongated along the ρ axis. For these reasons the algorithm which has been chosen is the DBSCAN [7].

DBSCAN algorithm [7] is based on cluster density with a parameter ϵ that represents the size of the neighbourhood of a point q, such neighbourhood can be described as follow $N_\epsilon(q) := \{p \in D | dist(p,q) \leq \epsilon\}$ and a parameter $minPts$ which is the minimum number of points in the given neighbourhood $N_\epsilon(q)$.

Since ϵ does not take care about scales, before applying the algorithm, the data has been rescaled in respect to its mean and variance in order to obtain 0 mean and unit variance.

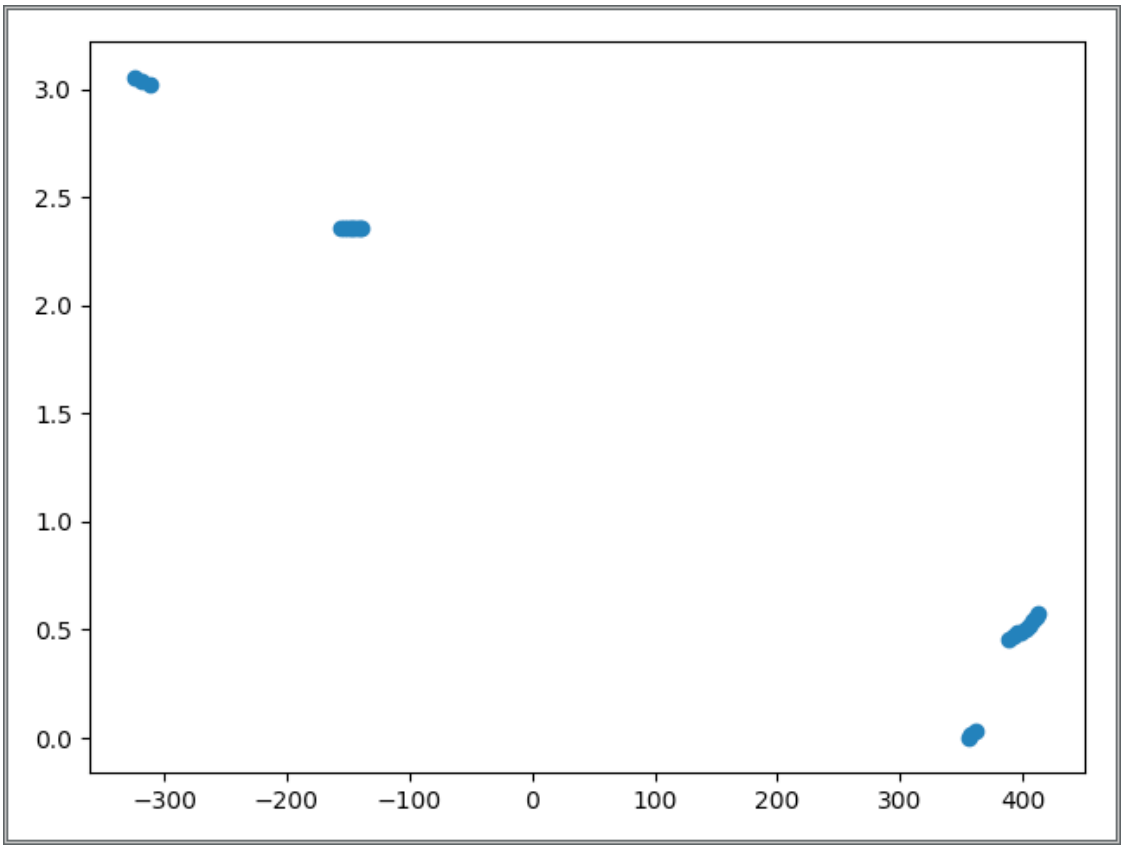


Figure 2.3: Example of lane detection with basic lane detector on $\rho - \theta$ plane

For the specific case the ϵ has been fixed to 0.1, and the $minPts$ has been set to 1 which means that no noise removal has been applied, so also a single hough lane is considered valid and the cluster resulted is itself.

In figure 2.5 the result of the clustering algorithm is shown, where each cluster is represented by a different colour, on the given image (figure 2.4), the lines that are candidate to be clustered are the blue ones with green inside them and the black ones are the results of the clustering.

Now there are candidate lanes, the algorithm considers the left and the right lanes such items whose intersections respect to the $view\_point$ (the orange line) it is closer to the centre of the image which is the $car\_position$ (the red circle in the middle of the image).

An example of the result of this approach is presented in the figures 2.6 and 2.7, where the identified left and right lanes are such lines with the red circle over them.

This approach has lots of limitations. The first limitation lies in the environment: the camera sees too much, we would like to have a reduced ROI to avoid negative influence of the environment. Another limitation is related to the fact that the front view of the camera does not allow to use some assumption on road lanes such

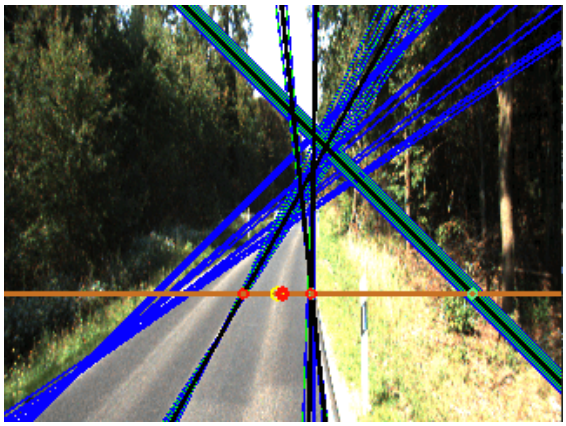


Figure 2.4: Source image

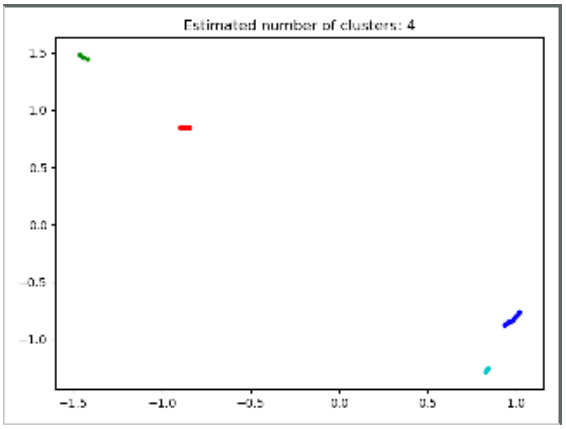


Figure 2.5: Clustering results on $\rho-\theta$ plane

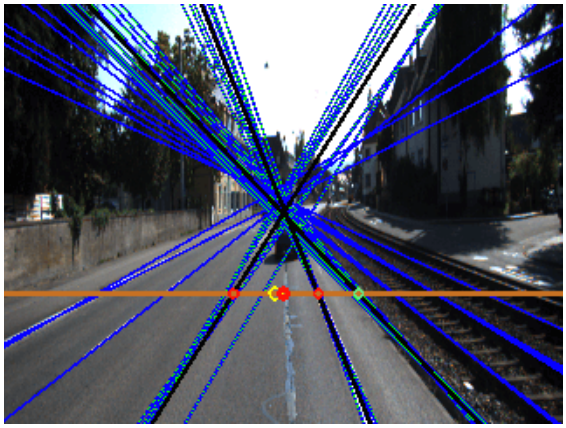


Figure 2.6: Example of lane detection with basic lane detector

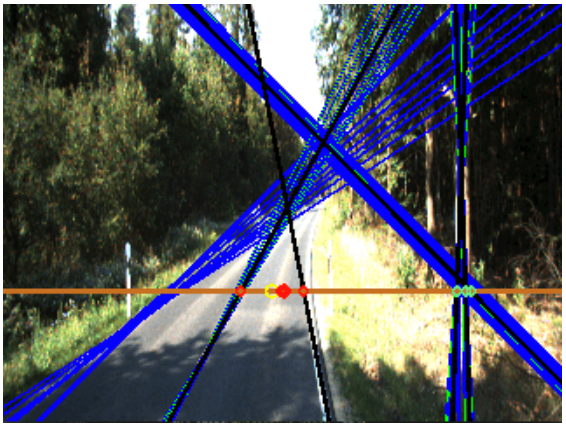


Figure 2.7: Example of line detection with basic lane detector

the parallelism.

Other problems are due to the strict approximation, this is not a good technique because experimentally we can find that sharp turns are not detected at all and dotted lanes have few votes to be strong candidates.

In addition this algorithm assumes that both left and right lanes are presented, but in some situations this is not true or the camera is not able to see both, common issue during sharp turns.

Shadows on road are still a problem because often the threshold is not able to remove them.

To overcame to all these problems except for the difficult lighting conditions the **Advanced lane detector** has been proposed.

2.2 Advanced lane detector

The general idea is the following: given a colour image, the goal is to apply filter operations in a manner that at the end of the process it will be possible to have a binary image where there will be only pixels of the lanes (see figures 2.8, 2.9). After this process, the idea is to search a second order polynomial that will fit these lanes.

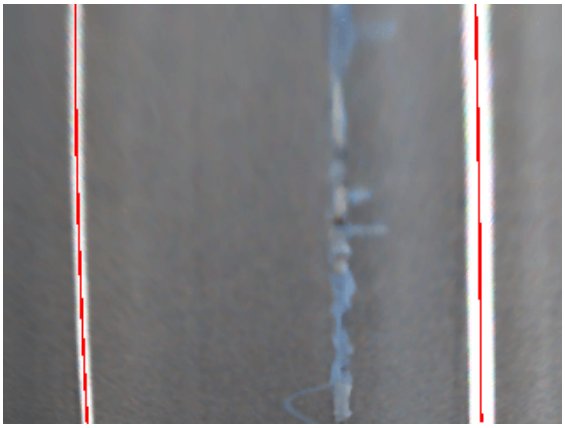


Figure 2.8: Example of road image with detected lanes

Figure 2.9: Result of the image after filtering

The algorithm pipeline:

- Frame preprocessing
  - Median Blur
  - Bird view
- Filtering
  - Convert into grey scale
  - Adaptive Threshold
  - Extract at most two lane origins
  - Create and apply ROI masks
  - Erode the image
- Fitting
  - Extract representative lanes pixels
  - Polynomial fitting on representative pixels
  - Checking quality of the fit

2.2.1 Frame preprocessing

Before starting the detection phase, some preprocessing operations are applied. In particular to remove image noise related to pixels with very different value in the BGR space respect to the neighbourhood pixels, a *median blur* is applied, this operation is a linear filter.

2.2.1.1 Median Blur

A linear filter operation consists on replacing each pixel of the image with a value called $g(i, j)$ which is determined as a weighted sum of input pixel values, the operation mathematically can be written as in 2.1

$$g(i, j) = \sum_{k,l} f(i + k, j + l)h(k, l) \tag{2.1}$$

$h(k, l)$ is called the kernel, which is nothing more than the coefficients of the filter.

The linear filters have a different name related to its operation. In particular *Median Filter* is a filter that returns as a result of the processed pixel the *median* of the pixel in the kernel.

In the algorithm the kernel has a size of 3x3 and it is applied at each image channel.

Noise reduction is very important also to obtain a good threshold.

2.2.1.2 Bird view

While driving the driver sees the road with a front point of view. In this scenario the eye or the camera of the autonomous car sees parallel lines intersecting to the horizon point. This situation is not good for detecting lines because in this configuration we lose some useful information such as the parallelism and the whole perspective transformation effect will be removed as mentioned in [1],[4] and [5] converting the front view into a bird view.

Another issue of the front view connected with the environment, since the camera does not see only the road but also the surrounding environment like sky, buildings and so on.

A better configuration to detect lines is to look starting from the sky to the road, as the birds do. The above configuration has the following advantages: the first one is that the camera sees the lines parallelism, so the algorithm can make more restrictive assumptions on location and shapes of the lines. Another useful

thing is the removal of the horizontal background from the image as the peripheral view, so it is possible to see only the road.

The goal is to convert the original image in a bird view image. To do that we need to assume that the road is flat and that the camera is inclined with a fixed angle respect to the road.

The common camera behaves as pinhole camera model, in this situation many of the transformations that may happen, can be described as a *projective transformation*.

This kind of transformations can be formulated as in 2.2.

$$g(\boldsymbol{x}) = f(s(\boldsymbol{x})) \tag{2.2}$$

where $s : \mathcal{R}^2 \to \mathcal{R}^2$ is a spatial transformation function, and $f : \mathcal{R}^2 \to \mathcal{R}^n$ is the image and the n is number of channels in such image. The s function can be described as the matrix multiplication in 2.3

$$s(\mathbf{x}) = s(x, y) = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.3}$$

H matrix is called **Homography matrix** and it has been calculated using the *getPerspectiveTransform* OpenCv function with 4 points of the first image and the related 4 points of the second image as inputs.

To perform the transformation the bilinear interpolation has been used (default in OpenCv *warpPerspective* function).

Assuming that the road is flat we can detect the perspective transformation, a rectangular sees in front view mode looks as a trapeze, so the goal is to find the function which maps the points of the trapeze into a rectangular.

To resolve this problem the OpenCv chessboard[1] has been used. In such a way as to map the boundary corners of the chessboard to corners of a rectangular. The result is something like shown in figures 2.10 and 2.11.

This operation permits to detect the homography matrix between the camera and the road so, if we apply such transformation to the original image, we will obtain the bird view.

Consequently the coordinates for the new point in the bird view images must be calculated has in 2.4, where λ is a scaling factor.

[1] download it from https://github.com/MarcoSignoretto/drAIver/blob/master/data/9x6_1-8cm_chessboard.png

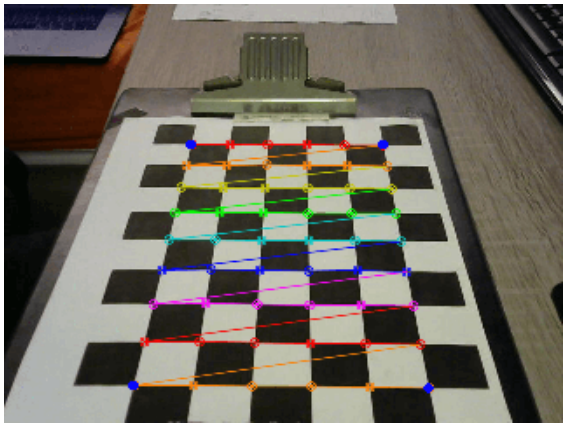


Figure 2.10: OpenCv chessboard front view

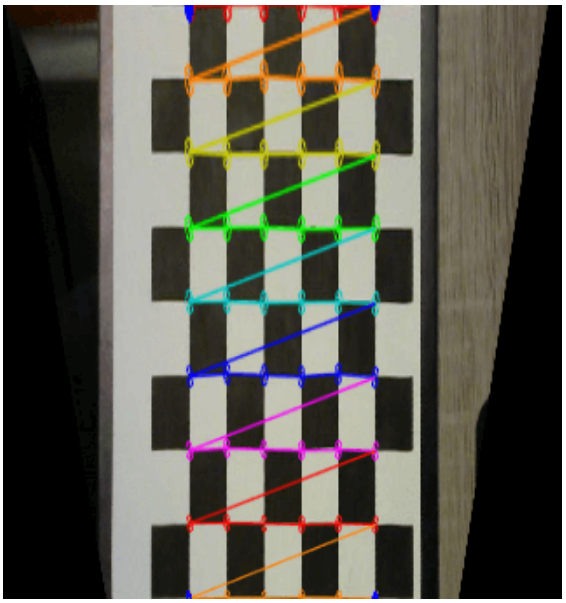


Figure 2.11: OpenCv chessboard bird view

$$H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ \lambda \end{bmatrix} => new\_point = \begin{bmatrix} \frac{x'}{\lambda} \\ \frac{y'}{\lambda} \\ \frac{\lambda}{\lambda} = 1 \end{bmatrix} \tag{2.4}$$

To sum up, this operation is very important because it allows to identify a better ROI overcoming the limitations of the first approach.

2.2.2 Filtering

In this section we will focus on the most important part of the algorithm.

The filtering operations are the core of the algorithm since their goal is to remove all parts of the image which are not lanes. This filtering could be improved by using deep learning approaches, but in this thesis the focus is to resolve this problem without deep learning approaches.

The filtering pipeline is the following:

- Binary threshold
- Erosion
- Sliding window mask filter

2.2.2.1 Binary threshold

Since the first approach shows that the difficult lighting conditions are a problem, in this one I tried to work directly on coloured images in order to avoid loss of colour information.

Instead of working on BGR colour space (OpenCv default) the HSV has been tried, but contrarily to expectations this approach was not so effective since white colour it is difficult to separate in such a configuration.

In HSV colour space we have:

H Hue or Dominant Wavelength

S Saturation or Purity

V Value or Intensity

The problem is that the white colour has not a range in the H channel because the white is composed by all the wavelength, so it is not possible to fix a range for the colour so that this filter is able to extract only white lanes. This issue increases in difficult lighting conditions, such as roads with shadows, since filter also on saturation and value becomes difficult to handle.

Therefore also in this approach the **adaptive mean binary threshold** has been used. This technique is effective also thanks to the bird view, which allows to remove the whole perspective transformation effect of the front view as mentioned in [5], where instead a Gaussian filter has been applied.

In this adaptive thresholding technique, the algorithm calculates the threshold for a small regions of the image. In this way we get different thresholds for different regions of the same image and it gives us better results for images with varying illumination [26].

The parameters of the adaptive threshold has been tuned manually and the final setup is $blockSize = 71$ and $C = -15$ which is a constant subtracted from the mean.

In the figures 2.12 and 2.13 there is an example of threshold operation.

2.2.2.2 Erosion

Respect to the approach that has been proposed on [1] and [5] a different one has been used to extract lines, and other filtering operations have been applied.

In the thresholded images there is often lots of noise. Since the lines are thick it is possible to remove some noise with erosion, in particular an erosion with a kernel size of 11x11 repeated 4 times has been performed.

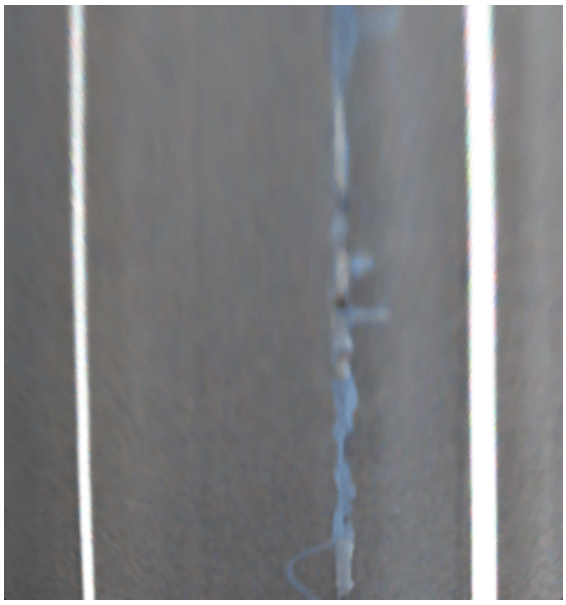


Figure 2.12: Original bird view image

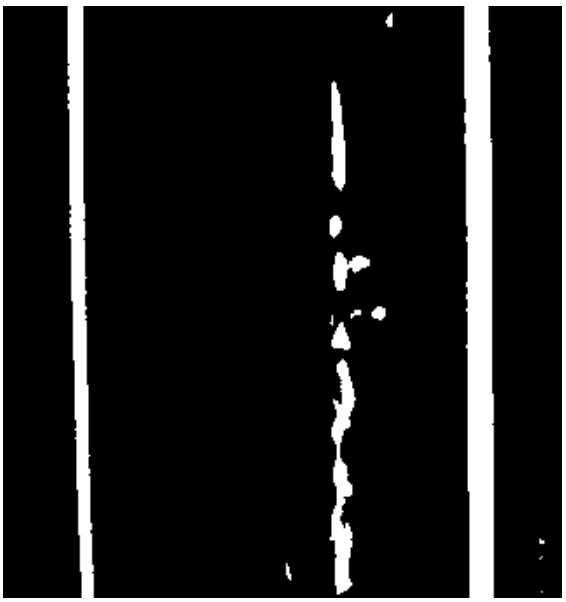


Figure 2.13: Bird view after threshold

This operation is very important to remove little noise and it allows the histogram stage to perform better, see next paragraph for further details.

Figures 2.14 and 2.15 show the bird view before and after the erosion operation.

Figure 2.14: Threshold on bird view image

Figure 2.15: Threshold on bird view image after erosion

2.2.2.3 Sliding window mask filter

In [4] the idea is to find Gaussian like shape to extract candidate points used to fit a polynomial, also in [1] has been used a similar approach to detect lanes using RANSAC spline fitting.

In our approach starting from the idea presented in [4] the goal is to obtain a mask, which is able to filter out part of the image that, with high probability, does not contain road lanes.

Base lane detection To start sliding window, we need to know where is the base line. To do so we make some important assumptions, which are the following: we assume that the lane is not to much sharp, the baseline of lanes are inside the width of the image and the car is positioned more or less at the centre of the road.

Under these assumptions an histogram will be computed. This one is the sum of pixel values over the height of the frame and normalized respect to the maximum value.

An example of such operation is visible in figures 2.16 and 2.17

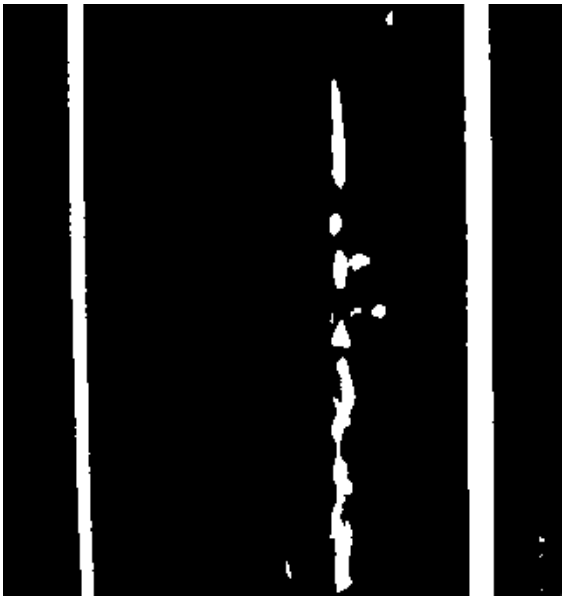


Figure 2.16: Source image

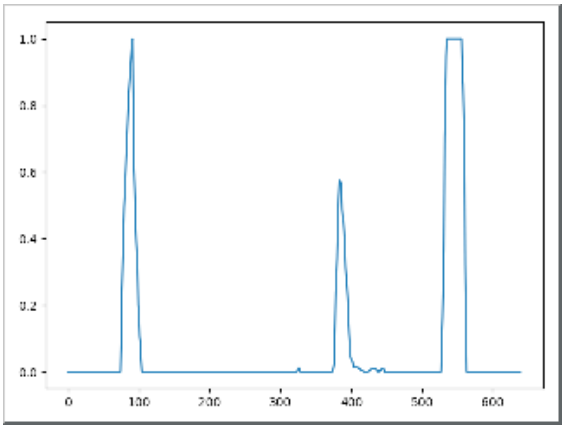


Figure 2.17: Baseline histogram

From this histogram the image is splitted at half of the width, and a maximum in the histogram will be searched on both sides, if the maximum value found is higher than the fixed threshold of 0.3, the line will be considered valid for the sliding window stage, otherwise we assume that such line does not exist.

In this scenario there is a problem when the robot or car approaches a very sharp turn, in fact in this situation it is not able to see both lines but it sees only

the far away line at the centre of the view as shown in figures 2.18 and 2.19. So the algorithm tends to find two lines very close together. To avoid this, a constraint on the minimum distance among lines has been added.

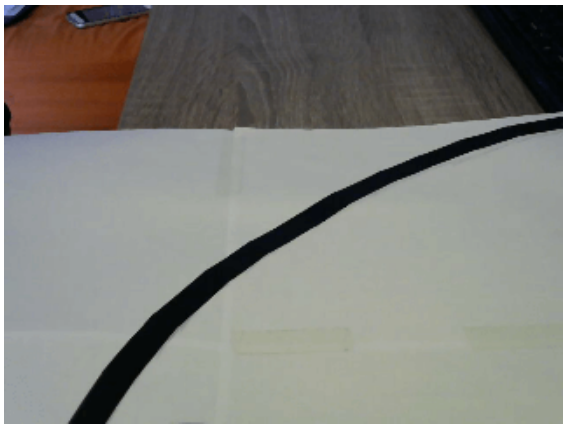


Figure 2.18: Very sharp turn front view

Figure 2.19: Very sharp turn bird view

This constraint is set to 200 pixels, if lanes are below such threshold, the two lanes will be detected as a single one and they will be classified left or right according to the higher value in the histogram.

This filtering part is very important, because it removes lot of noise and it allows to detect also single line overcoming the limitation of the first approach.

Sliding window Considered that we now have the base line positions for each lanes, the algorithm starts positioning a sliding window with its centre positioned in the middle of the window width.

From such window the algorithm computes the histogram inside this window and the peak is stored as the centre of the next window that starts from the top of the current one and it is positioned with the centre coordinates of the stored peak. This procedure is iterated until the top of the image, if during histogram computation there is not a peak, the algorithm will assume that it is between two pieces of dotted lanes.

In figures 2.20 and 2.21 there are two examples of the sliding window behaviour.

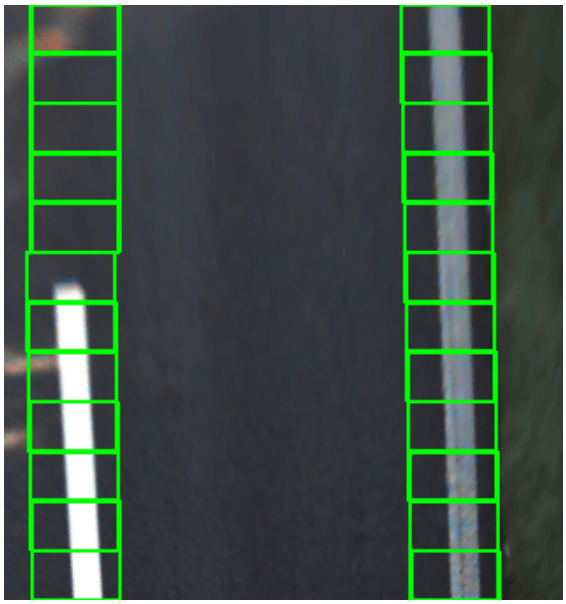


Figure 2.20: Sliding window method

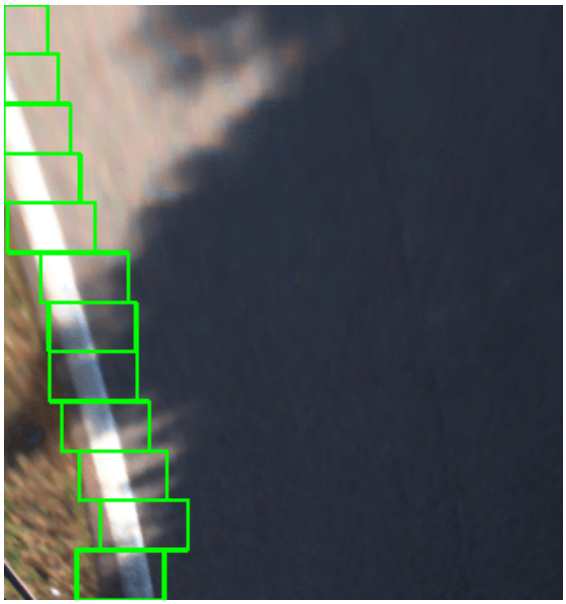


Figure 2.21: Sliding window method on curved line

This is a greedy approach, because the decision has been taken based on local information. This approach has some issues related to the sliding window movement, in difficult lighting conditions there are some problems because the peak in the window could be due to a medium size noise and this one could be move the window on wrong direction losing the correct lane trajectory completely. An example of such situation is visible on figure 2.22.

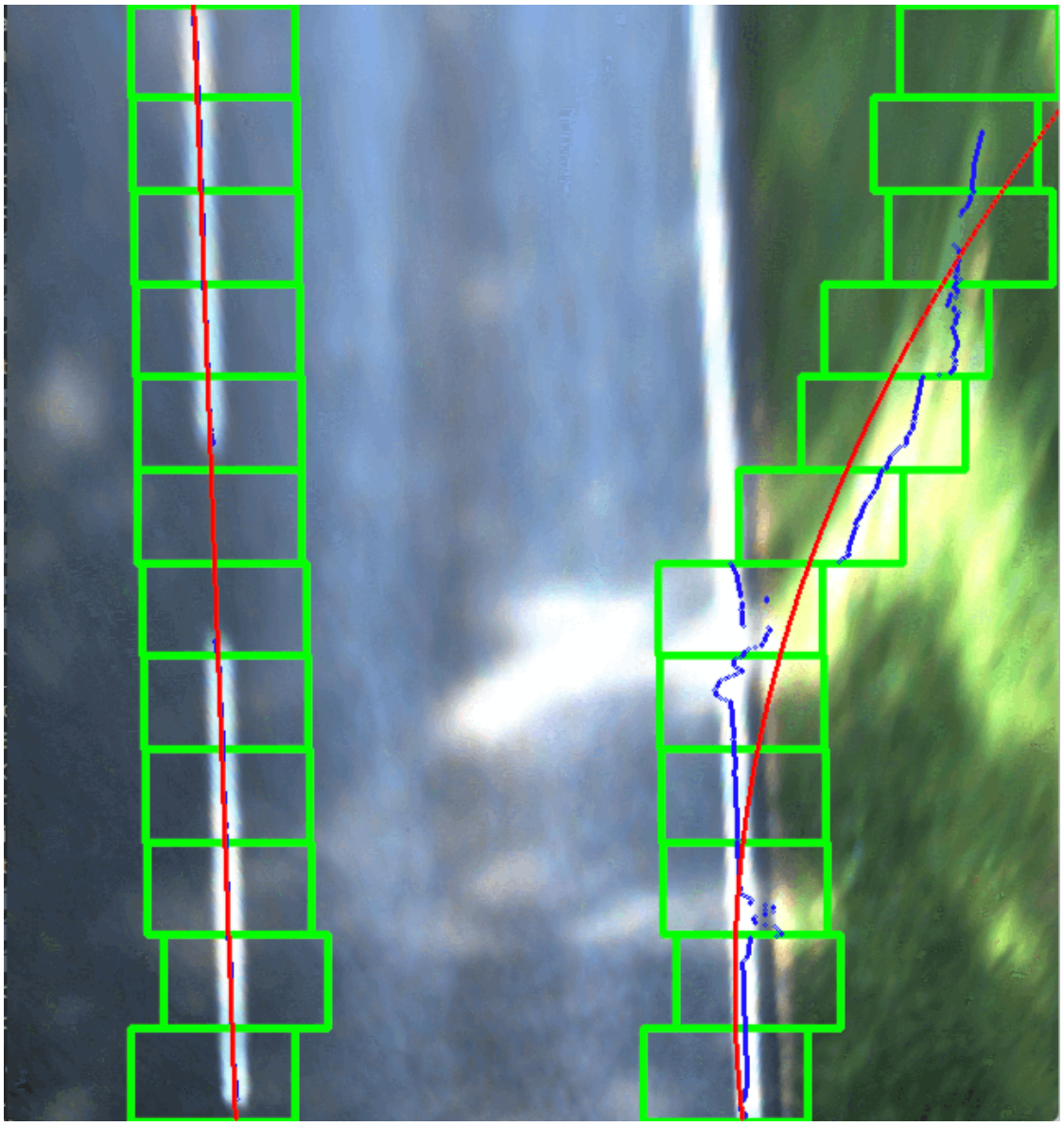


Figure 2.22: Sliding window problem

Chapter 8 will explain how this stage could be improved by using parallelism assumption.

The area occupied by the sliding windows will be considered as a mask of the processed line, the right and left mask will be applied to the threshold image, in this way we obtain a very effective filtering.

In figures 2.23 and 2.24 we can see how masking operation filters out lots of noise.

Figure 2.23: Threshold image before masking

Figure 2.24: Threshold image after masking

2.2.3 Polynomial fitting

In this **advanced line detector** the algorithm needs to fit a second order polynomial, consequently the algorithm is more robust and precise if compared to our previous proposed approach.

At this stage the noise should be always completely removed and the remaining part should not affect the result of the fitting so much.

2.2.3.1 Median points

In [5],[1] and [4] the fitting is based on few points to have a stronger and more reliable fitting; A different approach has been proposed.

The first step of the polynomial fitting consists on extracting, for each line, a pixel for each unit of the image height, this pixel should be as much as possible close to the line centre, the assumption that has been made during this process is that the remaining pixels of the image are only pixels of road lines or some little noise close to such lines.

Our approach to extract the centre of the lane is similar to the one proposed in [5] and [4]

The idea is to extract the pixel as median of pixels across the width dimension in the line mask. The median has been used instead of the mean because it allows to be less sensitive to outliers (noise).

The representative pixel is kept only if the number of pixels used to calculate

the median will be over the threshold of 30 pixels for robot and 10 for KITTI [14], otherwise it will be consider "unreliable" and discarded.

In figures 2.25 and 2.26 we can see in blue the median points.

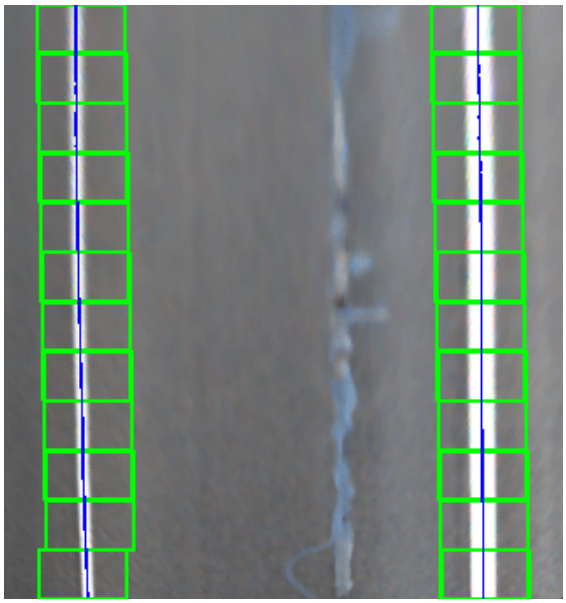


Figure 2.25: Median points in easy frame

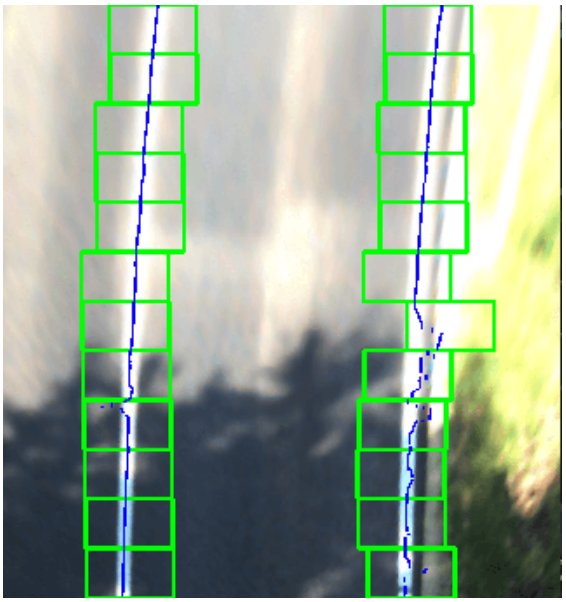


Figure 2.26: Median points in hard frame

IMPORTANT NOTICE: Since the code has been developed in Python the operations in $find\_median\_line$ are slow due to the for loop, which is present into the iteration procedure, so to reduce the number of items to iterate on, before applying the $find\_median\_line$ function, the thinning algorithm *Guo and Hall thinning algorithm*[18] has been applied to the threshold image. The algorithm implementation is written in C++ so it is very fast, this allows to reduce the iterations performed in Python. This approach has been only used to speed up the algorithm, but it loses precision and we obtain worst detections. For this reason the implementation of the $find\_median\_line$ function with C++ has been planned for future works.

2.2.3.2 Polynomial fitting

When the median points have been extracted, the algorithm performs a second order polynomial fitting under the conditions that the number of points available to the fitting procedure are above a threshold fixed to 100 points.

This process is a multiple linear regression of the model in 2.5, this model has been chosen, because it is able to model curved lanes and it is enough generic to avoid overfitting.

$$\mathbf{Y} = \beta_0 + \beta_1 \mathbf{X} + \beta_2 \mathbf{X}^2 \tag{2.5}$$

The above model can be described in matrix notation as in 2.6, where n is the number of points used for fitting. This conversion allows to perform such operation in parallel.

$$\mathbf{Y} = \mathbb{X}\mathbf{B} = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \dots & \dots & \dots \\ 1 & x_n & x_n^2 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} \tag{2.6}$$

we call $\mathbb{X}$ the *Vandermonde matrix* in 2.6 and $\mathbf{B}$ the vector of the beta coefficients. The estimation of $\mathbf{B}$ is performed using the *ordinary least squares*, so the $\hat{\mathbf{B}}$ is the estimator of $\mathbf{B}$ and it is equals to the equation in 2.7.

$$\hat{\mathbf{B}} = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}\mathbf{Y} \tag{2.7}$$

After polynomial parameters estimation, another filter has been applied: the numpy polynomial fitting allows us to look at the MSE, but it is not possible to look at the whole residuals so we cannot see if they have zero mean and are normally distributed, which is very important to understand whether the fit is good or not. Since we have only MSE information, I choose to divide that by the number of data points used for fitting in order to have a number that doe not depend on the number of data points. A threshold on the new error has been set, if the error is bellow 120 the fit is retained "good", otherwise it will be rejected. The threshold value has been found experimentally.

In figures 2.27 and 2.28 we can observe in red the representation of the inferred polynomials.

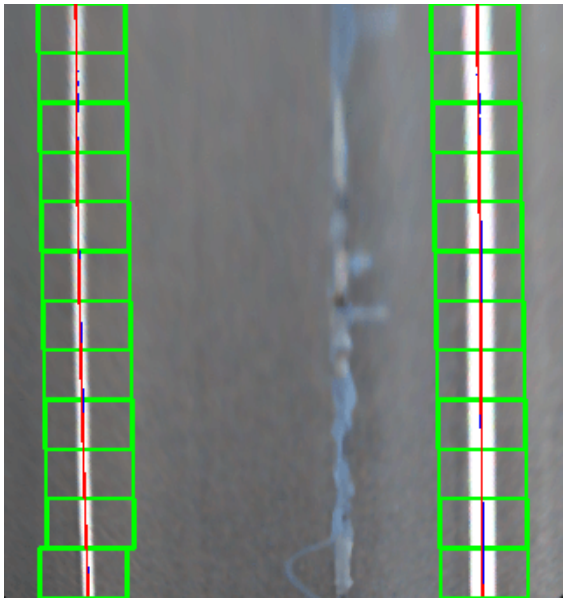


Figure 2.27: Polynomial fitting easy frame

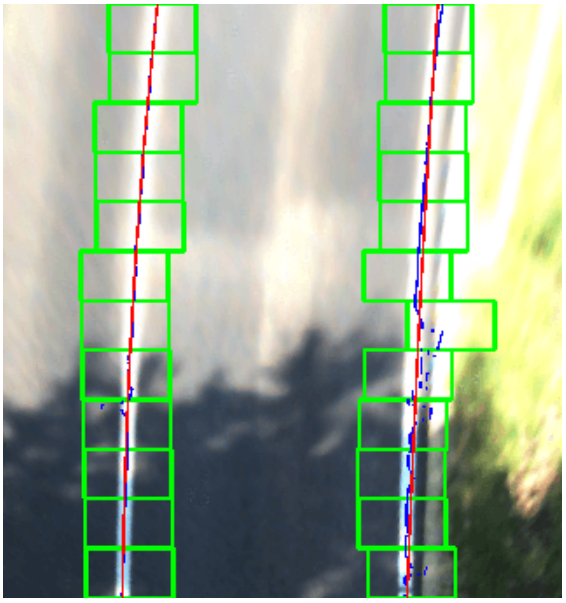


Figure 2.28: Polynomial fitting hard frame

Chapter 3

Object detection

Object detection is one of the most studied problem nowadays and lots of detector have been proposed, researches show that a good and effective way to handle this problem is using deep learning approaches based on CNN and for this reason this kind of supervised learning has been used in this project. For this type of tasks this approach is very effective. The problem is that these approaches requires lots of computational power to be executed, in particular we have to consider the real-time constraint that super impose certain bounds on processing time.

The goal is to find the correct architecture for the detection of pedestrians, cars cyclists and traffic signs, which allows to be fast enough to behave in real-time. For such reason we must accept a precision of the detection lower that the state of the art detectors.

3.1 Introduction to CNN

In the last few years the CNN have affected all the tasks which operates on images, this kind of neural networks have been show a great effectiveness for such tasks.

As described in [15], Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

3.1.1 Convolution

The convolution operation in two dimensions looks like in 3.1, where I is the 2D-input tensor (the 1 channel image), the K is the 2D-tensor representative of the kernel and the $*$ is the symbol used to describe the convolution operation. The flipped version of the original kernel (rotated by 180 degrees) is used during

convolution operation. Such operation with relative kernel flipping can be described with the formula in 3.1. This flip is important because allows convolution to be commutative.

$$S(x,y) = (I * K)(x,y) = \sum_{m=-p_w}^{p_w} \sum_{n=-p_h}^{p_h} I(x,y)K(x-m, y-n) \tag{3.1}$$

S is the pixel output at x, y coordinates of the image, the kernel size is $(2 * p_w + 1)$x$(2 * p_h + 1)$ and the kernel centre is at $0, 0$ so the negative coordinates for the kernel means up and left side respect to the centre.

The convolution operation allows **parameter sharing**, **equivariant representations**, which means that if the input changes, the output changes in the same way and **sparse interaction** that can be obtained using kernel much smaller in respect to the input image. In other words using kernel for convolutions means reducing the *receptive fields*, which is the number of neurons that fill neurons in the next layer; the reduction of the receptive field helps the network to avoid overfitting.

Graphically speaking, the convolution operation can be described as moving the kernel among the whole image, in figure 3.1 it is visible how the new pixel value is calculated, (notice that the original kernel is rotated by 180 degrees respect to the one in figure).

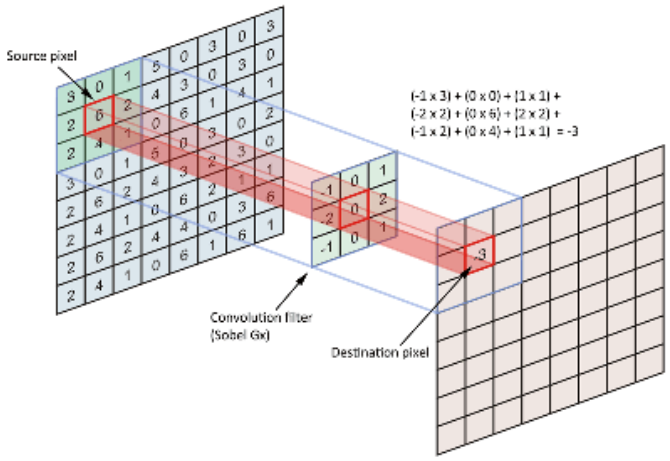


Figure 3.1: Example of convolution operation on a 2D-tensor

Often the result of a convolution operation is called **feature-map**.

3.1.2 Pooling

The second important operation that is used in a CNN is the pooling, this operation allows to reduce the size of the input tensor replacing the output of the net at a certain location with a summary statistic of the nearby outputs [15].

This operations is very important and allows to reduce the original tensor size to a size that will be used as input for a smaller convolutional layer, as feature for a normal neural network or as final output of the network.

There are several pooling functions, the most common is the *max pooling function*, which keeps the value with the higher value in a fixed window.

In this operation the concepts of **stride**, **size** and **padding**, is very important: the size is the width and the height of the window used during the operation, the stride instead is the number of pixels for which the kernel will be moved, the padding is another way to describe the size, $padding = \frac{size-1}{2}$ which means that it is the distance among kernel centre and it is far way cell of the kernel.

For example the convolution explained above uses a stride of 1 since for all the pixels of the original image there is a pixel on the output image. Also convolution operation can have a stride different from 1 but it's not so common.

Figure 3.2 reports an example of *max pooling* operation with a *kernel size* of 2x2 and a *stride* of 2.

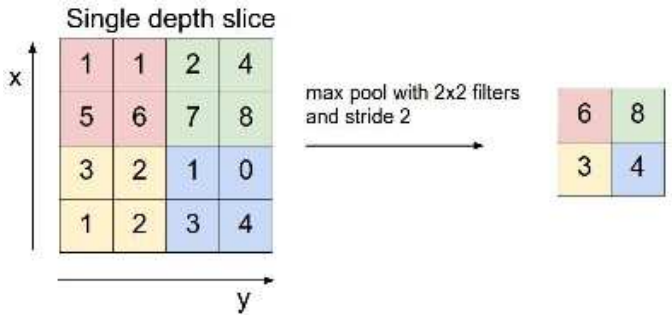


Figure 3.2: Example of max pooling on a 2D-tensor

Notice that the size of the output feature map is reduced by a factor of 2 which is the value of the *stride*, the stride should be considered as the downsample factor.

3.2 Network architecture choice

One of the most important constraint of the autonomous driving tasks is that they must be executed in real-time and with as much precision as possible.

So the goal is to find an object detector which is precise but fast enough to run on real-time. In order to make this choice the various aspects related to the the speed/memory/accuracy balance needs to be considered.

Nowadays there are several object detectors based on CNN, which have been implemented and studied. The most relevant are: R-CNN [32], SSD [24], YOLOv2

[29] and YOLOv3 [30]. On the paper **Speed/accuracy trade-offs** [20] the researchers have been compared such architectures except for YOLO v2 and v3, only first version of YOLO has been considered during the analysis.

In particular in [20] these networks are considered meta-architectures and these ones can be applied on different *feature extractors*, which are pure CNN architectures able to extract high level features from the images.

As a benchmark the authors used a very powerful machine with 32GB RAM, Intel Xeon E5-1650 v2 processor and Nvidia GeForce GTX Titan X GPU card.

Our project was born with the goal to run all the tasks on an embedding system, for us such system is a **Raspberry Pi 3 Model B** which is to limited to do so, but the **NVIDIA Jetson** could be able to run all these task on it (see section 6.3 for further details).

The two candidate architectures that have been considered and compared were SDD [24] based on the MobileNet feature extractor [19] and the tiny YOLOv3 architecture.

The performances are reported on table 3.1 and are evaluated on COCO dataset [23], the mAP has been kept from the paper [20] and from the official YOLOv3 website[1], the fps instead has been evaluated on a NVIDIA GTX 650Ti using the Tensorflow Object Detection API [16] for SSD with MobileNet feature extractor and Darknet [27] for the tiny YOLOv3 architecture.

Model	resolution	mAP	FPS
SSD MobileNet	256x256	18.8	27
Tiny YOLOv3	416x416	33.1	25

Table 3.1: Object detector comparison

Since the YOLOv3 has more or less the same speed of SSD MobileNet, but it archives higher mAP and it works on a higher resolution. Such architecture has been preferred. At the beginning we start with YOLO v2, in particular the *tiny* version of that, during project implementation the authors have released a new version of YOLO, which is YOLOv3 [30], since as explained below some problems have issued with the Darkflow [37] library which was based on YOLO v2 architecture, The library has been changed migrating to the original Darknet implementation [27] which is now based on the YOLOv3.

[1] https://pjreddie.com/darknet/yolo/

3.3 You Look Only Once version 2 (YOLOv2)

YOLO v2 [29] is one of the state of the art object detector.

As mentioned before we used the *tiny* version of YOLO v2 that in respect to the original *Darknet-19* architecture in 3.3 has the same structure as in table 3.2.

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

Figure 3.3: Original YOLOv2 architecture

3.4 You Look Only Once version 3 (YOLOv3)

In April 2018 the authors have released a new version of YOLO, which is the YOLO v3 [30]. This improved architecture version archives higher accuracy but a bit lower detection speed.

The project was started with the goal of training a tiny YOLOv2 architecture, but the Tensorflow library which provides this implementation does not work correctly, it had convergence and floating number approximation problems that make such library unusable.

For the above reasons the original library has been used. This one is not directly integrable with python, so the compatible version with Numpy and Python binding

#	Type	Filters	Size/Stride	Output
1	Convolutional	16	3 x 3	416 x 416
2	Maxpool		2 x 2/2	208x 208
3	Convolutional	32	3 x 3	208 x 208
4	Maxpool		2 x 2/2	104 x 104
5	Convolutional	64	3 x 3	104 x 104
6	Maxpool		2 x 2/2	52 x 52
7	Convolutional	128	3 x 3	52 x 52
8	Maxpool		2 x 2/2	26 x 26
9	Convolutional	256	3 x 3	26 x 26
10	Maxpool		2 x 2/2	13 x 13
11	Convolutional	512	3 x 3	13 x 13
12	Maxpool		2 x 2	13 x 13
13	Convolutional	1024	3 x 3	13 x 13
14	Convolutional	1024	3 x 3	13 x 13
15	Convolutional	260	1 x 1	13 x 13
16	Softmax			

Table 3.2: Tiny YOLOv2 architecture

support has been developed and it is available in my GitHub's fork **Darknet for drAIver**[2].

YOLOv3 compared to the YOLOv2 has some differences first of all the network architecture is based on *Darknet-53* described into figure 3.4.

The second main difference compared to YOLOv2 is that the new version predicts the bounding boxes and classes at different scales as other object detectors like SSD [24] already done. The original version predicts bounding boxes at 3 scales, on the contrary our tiny implementation uses only two scales.

The tiny version of YOLOv3 uses the same architecture of tiny YOLOv2 until the layer number 13, which is the end of the feature extraction part. To predict the detections at scale 13x13, after 13^{th} layer in 3.2, the YOLOv3 uses the architecture in 3.3. To detect at 26x26 scale, the output of the up-sampled result of the layer 13^{th} in 3.2 is concatenated with the output from the layer number 8 in 3.2 and such combination is used as input for the first layer of the architecture in 3.4.

The formula for filters written in the last layer will be explained in 3.4. Where the B_{13} is the number of bounding boxes for the 13x13 scale and the B_{26} is for the 26x26 scale.

[2]https://github.com/MarcoSignoretto/darknet

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 3.4: Original YOLOv3 architecture

#	Type	Filters	Size/Stride	Output
1	Convolutional	256	1 x 1	13 x 13
2	Convolutional	512	3 x 3	13 x 13
3	Convolutional	$B_{13} * (4 + 1 + C)$	1 x 1	13 x 13

Table 3.3: Tiny YOLOv3 architecture part for detection at 13x13 scale

#	Type	Filters	Size/Stride	Output
1	Convolutional	512	3 x 3	26 x 26
2	Convolutional	$B_{26} * (4 + 1 + C)$	1 x 1	26 x 26

Table 3.4: Tiny YOLOv3 architecture part for detection at 26x26 scale

3.4.1 How YOLO v3 works

YOLO network was projected to be fast and accurate. The first result was true from the first implementation of YOLO, instead as the accuracy of the first version of YOLO was under the other state of the art object detectors like SSD [24] and

[32] as explained in [20], with the introduction of YOLO v2 the accuracy grows up and it is comparable with the other fast detectors like SSD [24]. Now with the introduction of YOLO v3 [30] this accuracy has been improved one more time with a little drawback on the detection speed.

The idea of YOLOv2, which is the same of YOLO with some modifications, is to predict in a separate way the bounding boxes and the confidence for classes, in particular the system divides the input image into 13 x 13 grid. If the centre of an object falls into a grid cell, that grid cell will be responsible for detecting that object.

Each grid cell predicts B bounding boxes and for each one of them it predicts an objectness score and 4 bounding box offsets plus the conditional class probabilities $Pr(Class_i|Object)$ for each class of the dataset. Objectness scores reflect how confident is the model that the box contains an object and how accurate it thinks the box is, all this regardless what kind of object is present [29]. The B is related to the number of anchor boxes, and for each box it predicts the conditional class probabilities $Pr(Class_i|Object)$.

The location and dimension of the bounding boxes could be predicted directly but this configuration leads to unstable gradients during training, so most of the modern object detectors predict log-space transformations, or simply offsets to pre-defined default bounding boxes, called anchors [24], [30].

So YOLO predicts the bounding boxes as in 3.2 graphically represented in figure 3.5.

$$\begin{aligned} b_x &= \sigma(t_x) + c_x \\ b_y &= \sigma(t_y) + c_y \\ b_w &= p_w e^{t_w} \\ b_h &= p_h e^{t_h} \end{aligned} \tag{3.2}$$

b_x, b_y, b_w, b_h are the x,y centre coordinates, width and height of our prediction, t_x, t_y, t_w, t_h is network's outputs. c_x and c_y are the top-left coordinates of the grid, p_w and p_h are anchors dimensions for the box. σ is the sigmoid function and it is used to constraint the predicted offsets of x and y inside the cell as assumed in YOLO.

The bounding box confidence is defined as $Pr(Object) * IOU_{truth\_pred}$ and the class specific confidence score is calculated as $Pr(Class_i|Object)Pr(Object) * IOU_{truth\_pred}$

In [29] the anchor boxes have been calculated as the resulting centroids of the

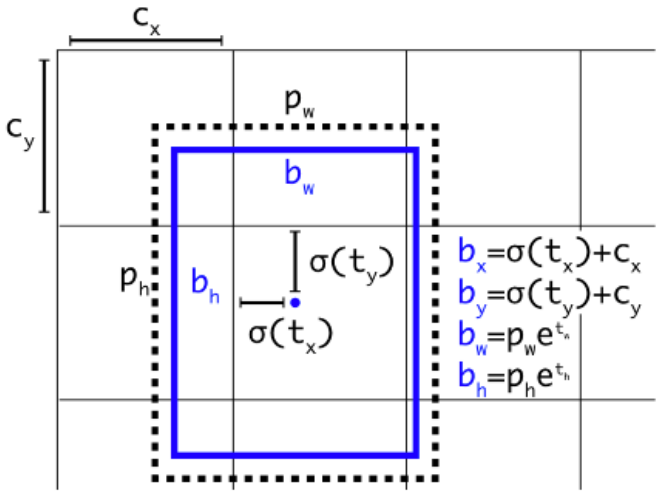


Figure 3.5: Bounding boxes with dimension priors and location prediction

K-Means algorithm [2] section 9.1.

You must remember that in the new YOLOv3 [30] an upsampling or deconvolutional layer with 2x scale factor has been introduced to support the detection at multiple resolution.

3.4.2 Training

To use such architecture, the *Darkflow* [37] library has been tried. Such library allows to use the original *cfg* and *weights* files available on the official YOLO website[3], but since this library has shown several problems during training phase with bad convergence and numerical errors, we used the original one, called Darknet [27] which implements the YOLOv3 architecture [30].

The YOLOv3 uses the *sum of squared error loss*, during the training phase, the input images have been manipulated to train the network in a more robust way, such transformations help the network to become invariant to some transformations and colour changes like in different lighting conditions, some scaling and cropping have been performed too, and example of such operations are available on figures.

In our case, we used the default network configuration except for the batch size during the training phase. Default parameters have been used because training procedure it is very time consuming and we have not time to try to improve the results with parameters tuning.

The initial weights of the network are the weights of the darknet53 pre-trained on ImageNet, so we have performed a fine tuning.

The training configuration is the following:

[3]https://pjreddie.com/darknet/yolov2/

Figure 3.6: Original Image

- **momentum term** $\alpha = 0.9$
- **learning rate** $\eta = 0.001$
- **learning rate decay** $= 0.0005$
- **decay steps** $= 400000$ and 45000
- **max batches to train** $= 500200$

The *momentum term* is the parameter used by the back propagation algorithm to mitigate the problem of a large learning rate η where the Δw_{pq} with the *momentum term* is calculated has show in 3.3

$$\Delta w_{pq}(t+1) = -\eta \frac{\partial}{\partial w_{pq}}(loss) + \alpha \Delta w_{pq}(t) \tag{3.3}$$

The *learning rate decay* is the number that has been subtracted by the *learning rate* at the step numbers listed in *decay steps* set, the algorithm iterates for *max batches to train*.

The batch size used during the training phase is 68 and each batch has been subdivided by 8 to fit into the GPU memory.

The training has been executed on a laptop equipped with a NVIDIA GTX 1070 and it takes 5 days for each network.

Figure 3.7: Modified Image 1

Figure 3.8: Modified Image 2

Figure 3.9: Gray Image

Figure 3.10: Cropped image

In order to adapt the networks to our tasks, starting from the *cfg*, the last final layer of each scales has been changed, according to what has been proposed on the official website [28]. The number of filters on the final layer is calculated as in 3.4, where 4 is related to the 4 bounding box regressions and the 1 is the objectness score.

$$anchor\_boxes * (4 + 1 + number\_of\_classes) \tag{3.4}$$

YOLOv2 [29] predicts bounding boxes and classes on the last convolutional layer, so to adapt the network for different classes we need to adjust such layer, in particular for the *Traffic Signs detector* over the LISA dataset [25], the number of filters in the last layer has been set to $260 = 5 * (5 + 47)$, for KITTI dataset [14] the number has been fixed to $70 = 5 * (5 + 9)$, note that the classes are 9 instead of 3 since in the dataset there is also a "DontCare" class that allows to avoid wrong detections and other classes with few examples on training set, so they are not considered. The number of filters on the last layer has been calculated taking into account that for each bounding box (anchor) we need to predict the x, y, w, h and

confidence for each class as explained in YOLO [31].

YOLOv3 introduces prediction at multiple scale, in our case the scales are two, which predict bounding boxes and classes at a feature map of 13x13 as in tiny YOLOv2 [29] and at 26x26. Since YOLOv3 [30] works at multiple scales, the anchor boxes have to be divided between such scales. We used the original configuration and we used 6 anchor boxes in total, 3 for each scale.

Darkflow's YOLOv2 implementation [37] uses the VOC annotation format, both dataset LISA [25] and KITTI [14] need to be converted.

To convert the annotations into the correct format two scripts have been required, the first one has been developed by myself and it is the script called `LISA_to_VOC.py`, which converts the LISA annotation format to the Darkflow one. The second script which converts the KITTI annotation format to the VOC one has been provided by the *voc-converter library*[4].

Since, as stated before, the Darkflow's YOLOv2 implementation does not work, the annotations have to be converted one more time for Darknet [27], In this way a new script has been realized to perform such operation, starting from the VOC format. This script is `VOC_to_darknet.py` and it has been used for both the datasets.

All training instructions are illustrated on the official *drAIver repository training section*[5]

The training process has been performed on a random 70% of the dataset, the remaining part has been used for performance evaluation. It was not possible to perform multiple trainings with different parameter settings, because there was no time to train the networks again, so no *evaluation set* has been used.

The same argumentation must be done for the anchor boxes proposal, so the VOC 2007[8]+2012 [9] k-means centroids proposed by the original article has been used as anchor boxes for the YOLOv2 cfg, the COCO's centroids have been used for the YOLOv3 cfg instead.

The *training set*, *test set* split for the dataset was performed using the `train_test_split.py` script. KITTI [14] has it own test set available for online submission, but since we want to evaluate the performances of LISA and KITTI in the same way we preferred to consider a part of the KITTI training dataset as *test set*, as it has been done for LISA.

[4] https://github.com/umautobots/vod-converter

[5] https://github.com/MarcoSignoretto/drAIver/blob/master/readme/OBJECT_DETECTOR_TRAINING.md

3.4.3 Detection

The trained networks have been used to detect the corresponding objects. These networks detect different objects, the confidence threshold has been set to 10%, this threshold should be higher in real world but it is appropriate for the simulated environment. For further details you should read the chapter 7.

The two networks works in parallel as explained in chapter **??** and for each frame they search objects in the resized image.

The robot frame has a different aspect ratio compared to the KITTI dataset. The first one has an aspect ratio of 4:3, while the second one of 414 : 125: this difference has a big impact on the detection phase on the robot environment, since the resized camera's frame has a different settings compared to the resized KITTI's frame, see chapter 7 to better understand what this imply.

In general, the detections perform well for both the simulated environment and the real one, LISA detection examples are reported in figures 3.11 and 3.12, for KITTI there is an example on image 3.13

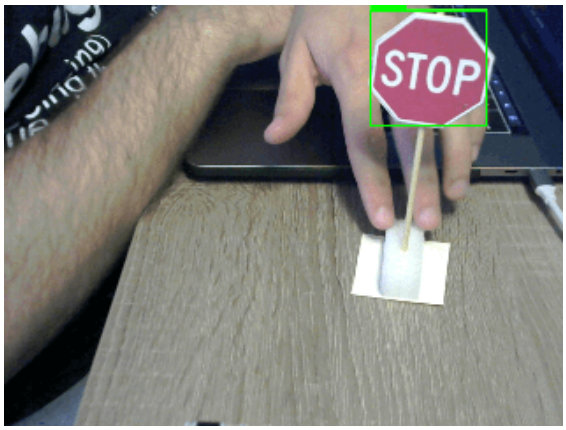


Figure 3.11: Stop detection example

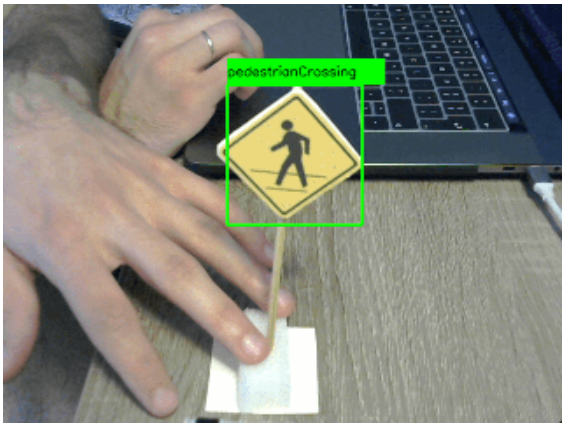


Figure 3.12: Pedestrian crossing detection example

What comes from the simulated environment experiments is that the proposed anchor boxes affect the results very much, you just need to move a little bit the object to pass from detection with high confidence to un-detection at all, this is where the YOLO architecture limitations issue.

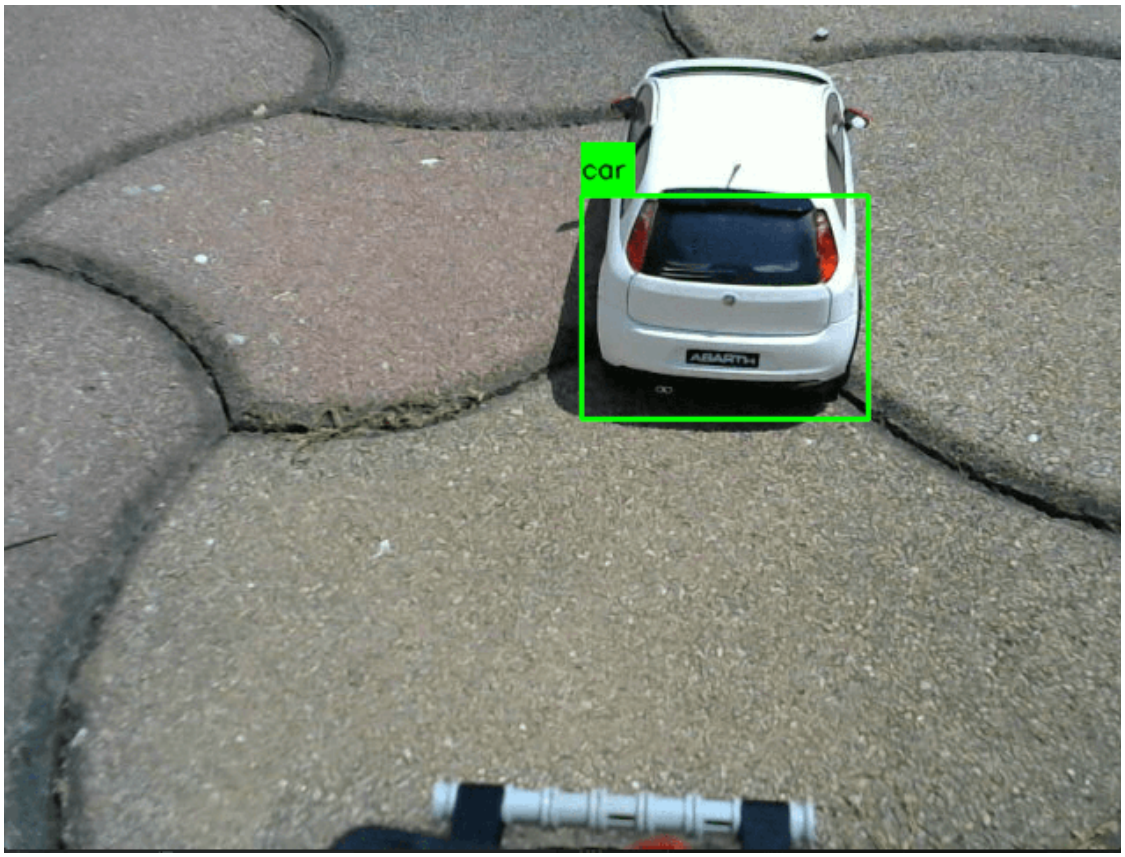


Figure 3.13: Car detection example

Chapter 4

Motion and Decision-making

In this chapter we will examine robot's motion and decision making on the simulated environment.

The robot has only a monocular camera as sensor, for this reason it is difficult to realize a robust and reliable navigation algorithm which requires more sensors (like GPS, lasers, multiple cameras and so on), to understand in a proper way the surrounding environment.

In this project also the decision making is simplified compared to the real-world, where there are lots of critical situations that in the simulated environment cannot happen.

We propose that further research may be undertaken in this issue with the introduction of end to end learning [3]. See chapter 8 for further information.

For our purposes the robot has to act in a simulated environment and in particular it must follow the road and behave correctly in case it meets a car or traffic sign.

To perform this task, the **Motion module** reads the **Driving state** with a fixed rate called $refreshing\_rate$ and uses this data as features to move, see section 5.4 for more information about the system architecture and how **Driving state** is updated.

The features provided by the **Driving state** are the following:

- last detection information about vehicles and pedestrians
- last detection information about traffic signs
- last left lane
- last right lane
- last base speed

- last steering information

These features are used to understand the whole situation and to take the correct action, such as to stop the robot whether it sees a car in front of it self in order to avoid any collisions.

4.1 Steering and throttling

The steering task is a very interesting topic and should be discussed in a separate work. In this project the steering task has been solved in a naive way, using a classical computer vision technique.

The steering approach, that has been used, is the same for both proposed line detectors that have been proposed, with some difference related the detection results.

Under the assumption that the car is more or less in the middle of the frame, the algorithm searches for the intersections with road lines (that could be also one or none) with a horizontal line which represents the $view\_point$ of the car.

If the robot is able to detect one or two lines, it will move. In particular there is a base speed that we called $motion\_speed$: this one is applied on both robot's motors, the steering procedure is related to speed adjustment of each single motor according to the distance between the centre of the frame ($car\_position$) and the centre of the road, which is calculated as the distance between the two road lines. Furthermore if only one line is detected, there will be a road width assumption of 300 pixels, in such situation the missing line will be assumed to be 300 pixels far away. The subsection 2.2.2.3 explains how the robot chooses if the single line is the left one or the right one.

The width assumption has been used only on the **advanced line detector** and it works very well on the simulated environment.

In figures 4.1 we can see the relevant points for steering, in purple there are the road line intersections, in blue the car position and in orange the road centre.

The steering mechanism is very simple, the idea is to calculate the car offset as the distance between the $car\_position$ and the $road\_centre$, the $delta\_speed$ is calculated as show in 4.1.

$$delta\_speed = \frac{STEERING\_SCALE * car\_offset}{steering\_range/2.0} \tag{4.1}$$

where $steering\_range$ is the distance between left and right road lanes and the $STEERING\_SCALE$ is a constant used to adjust the steering radius.

Figure 4.1: Threshold image before masking

delta_speed is summed to the *motion_speed* of the right motor and subtracted from the left one.

When robot approaches a turn, the *motion_speed* is decreased by a factor of $\frac{motion\_speed}{2}$.

In this way the robot is able to follow the road in a simulated environment and when no lanes are detected it will stop.

4.2 Understanding obstacles and avoid collisions

Until now we have had a robot which is able to follow the road but it is not able to understand the obstacles yet.

This problem has been solved by using the detections of the *last detection informations about vehicles and pedestrians* which are the pieces of information provided by the network trained on KITTI [14] which has been discussed in chapter 3.

The detection identifies a bounding box on a $front\_view$ scenario, this makes evident a problem: the obstacle distance estimation.

Since the robot has not sensors, that allow it to detect distances, we proposed a distance estimation based on Hough transform as proposed also in [39] and [40].

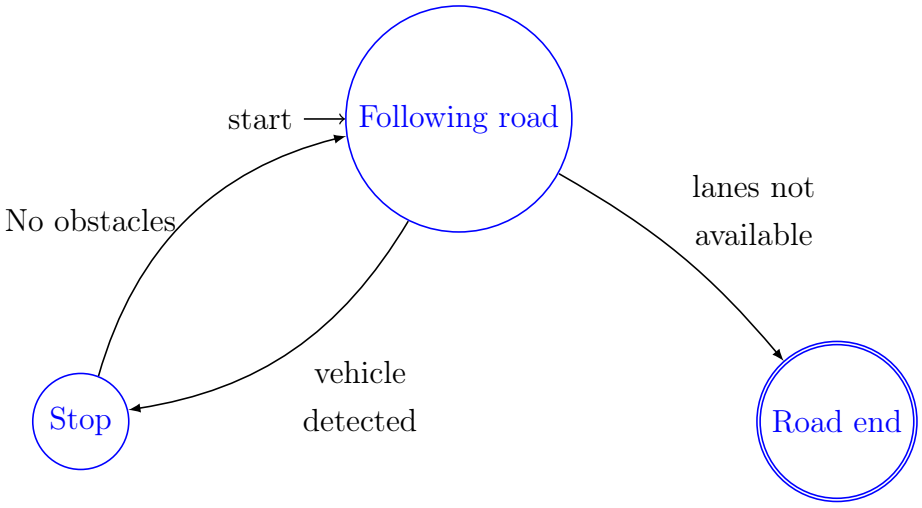
For us is easy to calculate such estimation, since we already know the *Homography matrix*, calculated during the **birdview** transformation, see section 2.2.1.2. So to estimate the distance of the obstacle from the robot it is enough to compute the operation in 2.4 (where x, y are the coordinates of the *bottomright* corner of the detected bounding box) to obtain the coordinates in the birdview plane, a scenario where the perspective transformation effects on road distances have been removed , and so the estimated distance from the object in pixels can be calculated as in 4.2.

$$birdview\_image\_height - \frac{y'}{\lambda} \tag{4.2}$$

With this distance information the robot must take decisions and manage some priorities.

Robot's task behaviours are modelled as deterministic finite automata as explained in [13]. The automata allows to model the various events as transitions in the automata, the presence of these states allows us to have a natural priority of some events compared to others.

In graph 4.2 we see the basic states of the system and the transactions related to the vehicle detection events. As we can note if the system is in "follows road" and a vehicle is detected, the state will commute to "Stop". This behaviour implies that the "vehicle detected" event has a higher priority compared to the "lanes available": this is our goal.



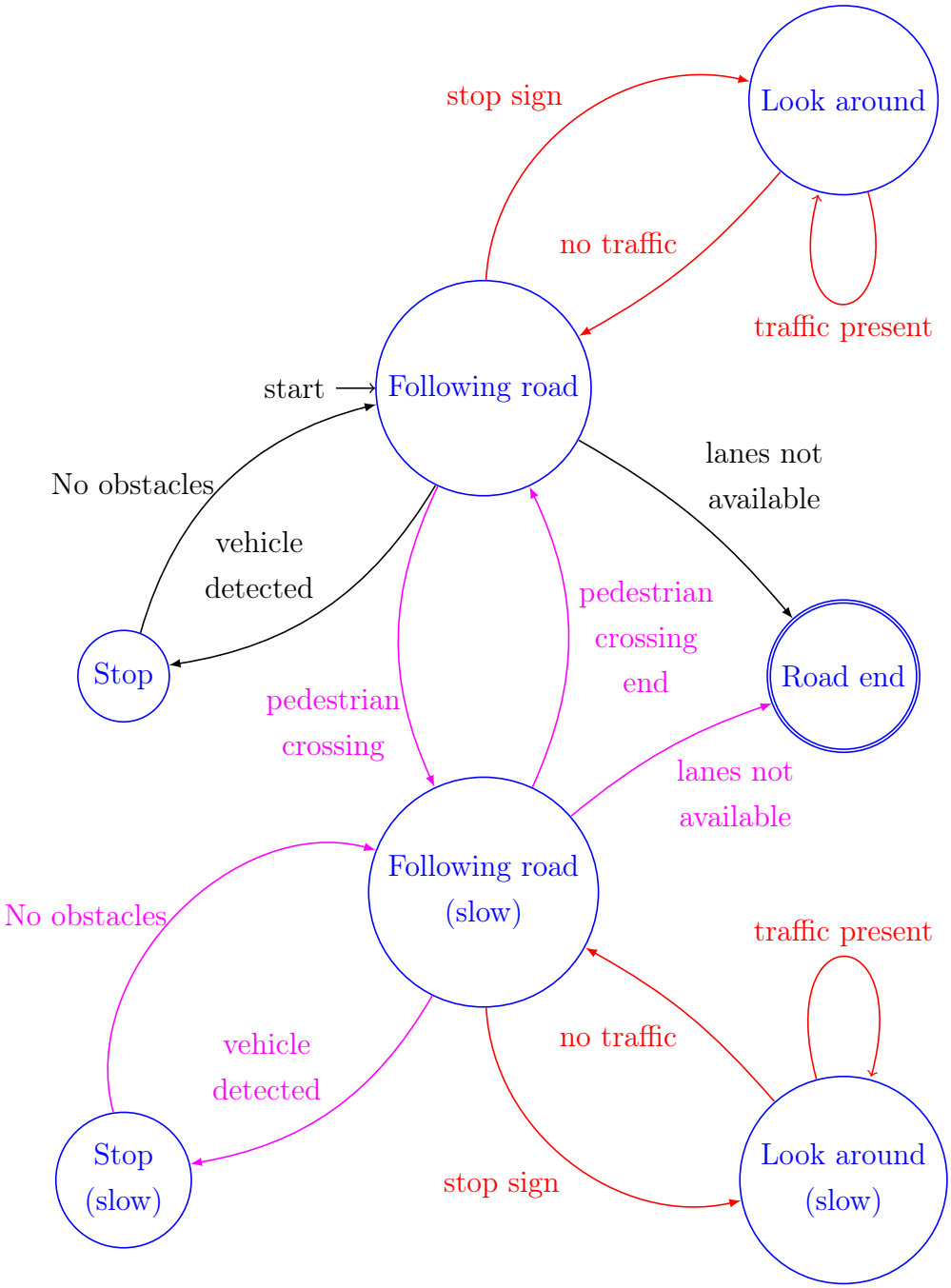
For robustness reasons, the "No obstacles" event fires only if there is not obstacle detection for at least 50 consecutive time intervals (a time interval is $\frac{1}{refreshing\_rate}$), this allows the robot to be very sure and safe when it starts to run again.

4.3 Traffic signs decision making

The traffic sign detection and related actions are more difficult in comparison to *vehicles detection*, where the only possible action is to stop the robot.

For the the second kind of detections should be associated a different action for each traffic sign and these actions must have a lower priority compared to the vehicle detection but higher that road following.

If we consider the graph in 4.2 it can be extended by adding the new states related to the traffic sign's actions. These actions, supported by the robot, are related to the events: 'stop traffic sign detection' and 'pedestrian crossing traffic sign detection', the resulting graph is reported in 4.3



In red have been reported the transitions related to the "stop traffic sign detection", in magenta the ones related to the "pedestrian crossing traffic sign detection".

Each action can be added to the graph adding the relative states and transitions so this model is able grow up.

As we can see from the finite state automata, this one respects all the priority policies that we want to grant so the "vehicle detection" has maximum priority then there is the *traffic sign detection events* like "pedestrian crossing" and "stop sign". Each event has an higher priority respect to the behaviour of lane following.

Chapter 5

Software architecture

In this chapter we present the communication architecture among robot and computer and the internal modules data flow. The main goal is to reduce as much as possible the communication time among robot and computer and between modules.

The system needs a bi-directional communication, for simplicity we call the communication from the robot to the computer *robot to computer* and the opposite *computer to robot*.

The first choice that has been made to reach a good transmission speed is the use of a *point-to-point* communication, which allows to remove the overhead that an external access point introduces. In this way if the computer is close to the robot, we will avoid problems of signal degradation with relative speed dropping.

The robot provides an access point as mentioned in section 6.2.2, this access point is configured to resolve a DNS name, which is *drAIver.local*, in this way it is easy to establish the connection between computer and robot, because we can use the DNS name.

5.1 Robot to computer

The robot's goal is to send data images to the computer as fast as possible.

The robot starts to capture camera frames and send them to the computer over the network. This operation is performed by sending first the image size with a fixed size packet and then the entire serialized image.

The communication protocol is TCP. At the beginning I tried to use the UDP protocol with the idea that if a frame is lost, it will not be a problem. This is true, but the problem comes from the fact that the image size is to large to be sent in a single UDP packet, so reconstructing the image with lost packets becomes difficult. Consequently we use the TCP protocol, which integrates the error checking and it

ordered the packets in the correct way, as explained in [36] section 1.4.4.

Sending the raw image is a bad choice, because it occupies a very big size in memory, that means lots of data to be sent. To avoid this problem and maintain fast the communication the images are first compressed in JPEG format using quality of 100 and then sent through the network.

JPEG is a **lossy compression** so some information of the original image will be lost, but experimentally we can say that this does not affect the quality of the task results, but the transmission speed is 4 times faster: from an average of 0.24 seconds for image to an average of 0.06 seconds for image.

5.2 Computer to robot

The goal of the computer is to send commands to the robot; as before the robot waits that computer connects to it and then it starts to send commands.

The communication protocol is based in TCP protocol as before, each message has a fixed length of 2 bytes, 1 for the left motor speed and 1 for the right one, the most significative byte is related to the motor left.

Each byte represents an integer, the value of the number is related to the motor speed, the motor speed range is between 100 and -100. Since the **BrickPi3** motor library considers only the last significative byte for speed, if the integer constructed from the received message is not negative, because the most unused significative bytes are zero, the hardware works as well.

5.3 Wireless issues

Considering that the communication is wireless and bi-directional the bottleneck is the collision of the packets over the air. So a good choice is to choose a wireless channel which is the most clean; this can be done monitoring the area wireless with smartphone applications like **WIFI Analyzer**. A typical mistake is to consider also the bidirectional communication source of conflicts, but this is not our case, because we are in a situation like in figure 5.1.

In this situation both devices before starting communication listen to the channel and since the devices are only two it is always possible to avoid any conflicts. As mentioned before, the problem are the external sources. For example if we have a situation like in figure 5.2 and we suppose that external source is transmitting on the same channel of robot and computer and the computer wants to send a message, in this case the computer will listen at the channel and found it free,

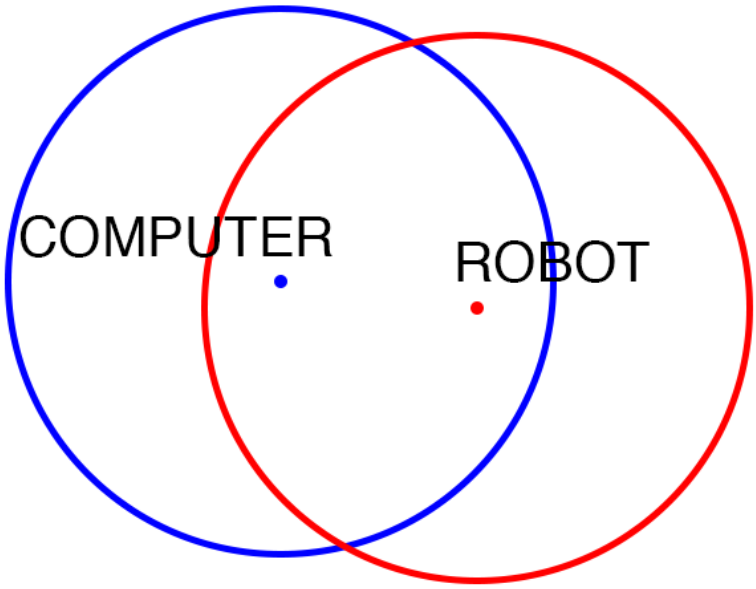


Figure 5.1: Robot and computer wireless channels

because the external source signal cannot be captured by computer, because this one is far away. So it starts its communication, but robot that sees both, receives a corrupted communication.

For further details about the problem explained above see [36] section 4.4.3 Protocol MAC 802.11.

5.4 System architecture

The system has been modelled has and asynchronous one. This one is composed by different modules. The whole architecture has been inspired from the one proposed in [13]. The most important are the following:

- Lane Detector module
- Traffic Sign Detector module
- Car Detector module
- Camera module
- Motion module

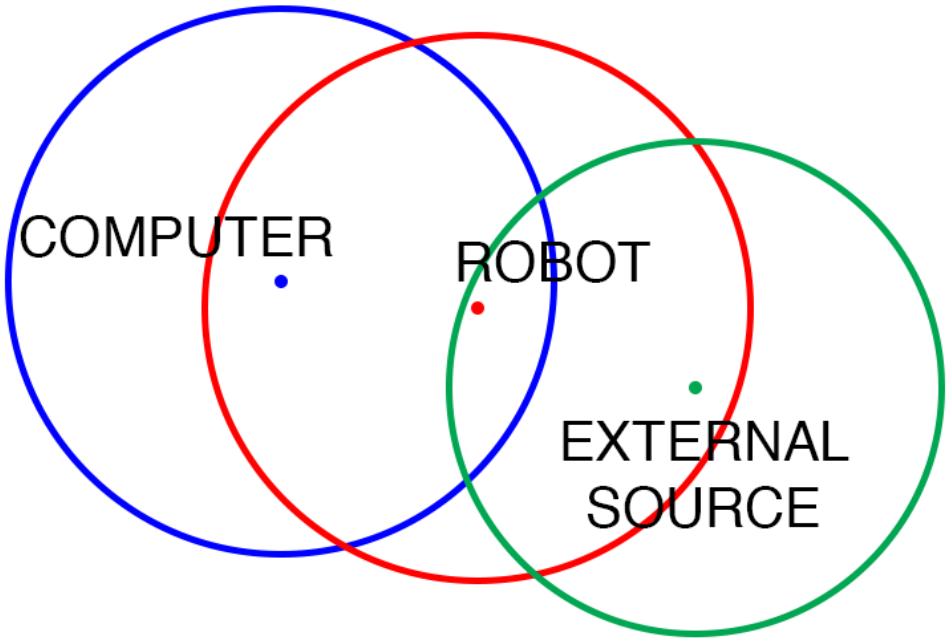


Figure 5.2: Robot and computer wireless channels with external source

All these modules needs to share information, for this reason we have taken in consideration the ROS [11] operating system, where the modules, called nodes, communicate through a channel, called topic.

One of the most important topic of this section is about how modelling the World. This is very important and it affects the modules which require these pieces of information. This model has been implemented in Object Oriented fashion, as proposed in [12]. In particular we had to chose among two solutions, related to how notify the correct real-time world state to the module responsible of decision making. The first method is to use the *Observer pattern* when the module interested in the world model will be notified when there is an update, the second solution is that the decision making module is responsible for asking the state to the model when it needs it. We preferred the second solution, because the real-time model is updated by multiple sources and it changes fast, so to avoid too many communications from the robot to the computer the new state has been polled with a fixed rate, as explained below.

The robot has a global state called *Driving state* that represents the *World model* described in [12] where the entire information is collected and updated. Since the robots needs to have as recent as possible information, a special kind of Queue has been implemented. This kind of queue is called `SkipQueue` and it keeps only the

newest information. In this way if there are some delays during the communications between modules or between computer and robot, the system will be sure to have the most recent data always available.

This queue is very important since it allows the system to behave in real time, the queue skips older frames, if the elaboration is not enough fast to process the next frame before a new one arrives. In this way we avoid to increase the request queue.

The system architecture of the computer part is visible in figure 5.3. The camera frame was received over the network, the *Frame provider* module convert it into an OpenCv image and it sends it over the four main modules which are *Traffic sign detector module*, *Lane detector module*, *Cars detector module*, *Rendering module*. These models execute their tasks in parallel and when they have finished they update the *Driving State* in an asynchronous way. The *Motion module* polls the state from the *Driving State* with a fixed rate and send such state through the network at the robot that acts properly. The script that uses this architecture and performs all these tasks is the script located into `scripts/drAIverComputer_v3.py`.

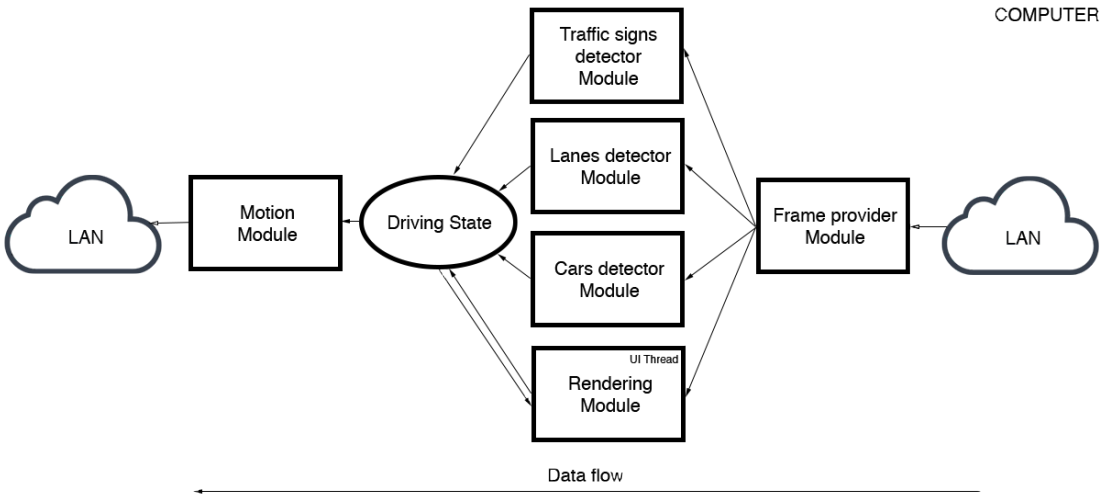


Figure 5.3: Computer architecture

The robot architecture is very simple and allows the robot to act as a rover. The robot collects as fast as possible frames from the *Camera module* and put the most recent frame into a `SkipQueue`, then the *Sending module* collects it from the queue and send it in a blocking way through the network.

On the other hand the robot receives commands from the computer through the network and propagates them to the motors thanks to the *Motion module*, such commands are time bounded, so if the communication speed is not enough fast to ensure fresh data for the robot, it will stop and it will wait the next command

before proceeding. This mechanism is fundamental to handle delays in a proper way, otherwise the robot will lose the road at the first delay.

The explained architecture is visible in figure 5.4 and the script that uses this architecture is located at `scripts/drAIverRobot.py`.

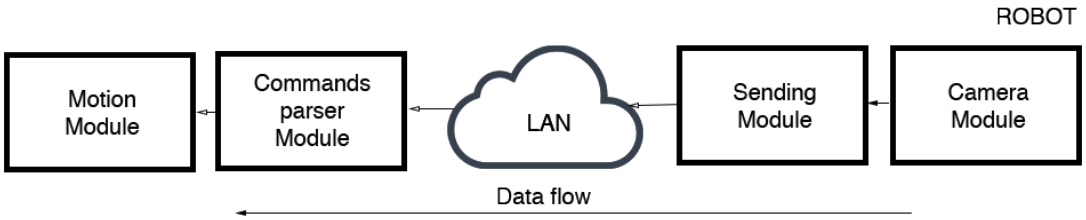


Figure 5.4: Robot architecture

Chapter 6

Hardware and environment setup

6.1 PC environment setup

In this chapter I will explain how to prepare the desktop or laptop to run the core part of the robot.

Since robot uses deep learning methods and Computer Vision methods, the main libraries used are **Darknet** [27] and **OpenCv3**[26].

Tensorflow [17] has been used during development but it hasn't been used in final code since it doesn't work correctly in the YOLOv2 Darkflow library [37].

The above libraries, in particular **Darknet**, have the capabilities to run on NVIDIA CUDA GPUs, in this way we can speed up the heavy computations (high speed up for deep learning tasks).

The version of **Tensorflow** that I used is the 1.4.0 and the version of **OpenCv3** is the 3.3.0, for **Darknet** instead, MarcoSignoretto's GitHUb fork has been used.

6.1.1 NVIDIA CUDA

In order to have good performance and be realtime we need to setup the NVIDIA CUDA Toolkit and NVIDIA cuDNN library.

Since I used the Tensorflow 1.4.0 I need to use the *NVIDIA CUDA Toolkit 8.0* and the *NVIDIA cuDNN 6.0* because this version of Tensorflow support only this version of CUDA.

This CUDA version is compatible also with **Darknet**.

All the technical procedure and installation instructions can be found at `https://github.com/MarcoSignoretto/drAIver/blob/master/readme/ENVIRONMENT_SETUP.md` under the **NVIDIA Cuda** section.

6.1.2 Anaconda3

Anaconda is a powerful tool useful to create python virtual environments.

To start with, I setup the environment using python virtual env but in this way I needed to compile for each computer the OpenCv library and installed all the dependencies manually.

At a certain point I discovered Anaconda3 that allows me to create an environment file to specify all the virtual env dependencies, and thanks to the **AnacondaCloud**[1] it is possible to specify dependencies of lots of libraries and then anaconda downloads the correct compiled library for the correct operating system we are using. This is very useful and it allows an easier setup on different computers.

Some warnings are related to the fact that some libraries are not available for all the operating systems, for this reason I created different environment files for the different operating systems.

Since after a while windows operating system became difficult to support, I have chosen to dismiss the support for such operating system, so drAIver project is compatible with Ubuntu 16.04 and macOS High Sierra.

Also for the setup using Anaconda there are the technical procedure and installation instructions at https://github.com/MarcoSignoretto/drAIver/blob/master/readme/ENVIRONMENT_SETUP.md under the **Anaconda** section.

6.1.3 Darkflow

Darkflow is a library which allows to use Darknet[2] networks inside tensorflow.

This library is required because the network for YOLOv2 [29] has been developed using Darknet but our project is based on Tensorflow [17] which allows to run the same network on CPU and GPU making easier the development on computers without GPU capabilities.

To configure this library you have to refer to the *computer environment setup instructions*[3] under the **Darkflow** section.

Unfortunately this library does not work properly and during training phase the library fails, for this reason we have changed the library and we have used the original one which is *Darknet*

[1] https://anaconda.org/

[2] https://pjreddie.com/darknet/

[3] https://github.com/MarcoSignoretto/drAIver/blob/master/readme/ENVIRONMENT_SETUP.md

6.1.4 Darknet for drAIver

The original Darknet library is not directly integrable with our python code, for such reason instead of the original one it's important to use our forked version **Darknet for drAIver**[4].

To configure this library you have to refers to the *computer environment setup instructions*[5] under the **Darknet for drAIver** section.

The main difference between the original version and the forked one is that the second one has been modified to support Numpy arrays as images which is the format used by OpenCV library.

Respect to **Darkflow** [37], **Darknet** needs to be compiled on each computer since it is written in C++ code.

6.2 Robot environment setup

The robot setup is quite complex, for this reason a detailed technical guide is available on GitHub at RobotSetup[6]

For an easy installation, on the file mentioned above there is a link to download a pre-setupped image of Robot's system.

6.2.1 Operating System

First of all we need to install the operating system on the RaspberryPi, for this purpose instead of install the common RaspberryPi Operating system called *Raspbian* which is based on Debian linux distribution, I have used a modified version of it which is called Raspbian for Robots.

This operating system has installed some facilities and optimization useful for robot applications, this operating system is supported by *Dexter Industries* [21], the same people that developed the BrickPi board and library.

To install the operating system see the section **Installing Raspbian for Robots**

6.2.2 Hotspot

In order to communicate with the PC the robot provides a wireless hotspot, in this way the computer can connect to the robot and communicate in a point to

[4]https://github.com/MarcoSignoretto/darknet
[5]https://github.com/MarcoSignoretto/drAIver/blob/master/readme/ENVIRONMENT_SETUP.md
[6]https://github.com/MarcoSignoretto/drAIver/blob/master/readme/ROBOT_SETUP.md

point fashion. The communication detail will be explained on chapter 5, also this setup steps are explained in section **Hotspot setup** of the robot setup page on GitHub.

6.2.3 Robot virtual environment

Robot virtual environment is based on a python virtual environment called **drAIver**, and on this environment all the required libraries have been installed.

6.2.4 BrickPi3 library

As mention before this is the library provided by *Dexter Industries* which provides all the python interfaces useful to control the LEGO's motors and sensors.

6.2.5 OpenCv3

Compared to the pc environment, the installation of the OpenCv library is more complex, OpenCv doesn't provide a compiled version of the library for the RaspberryPi and there isn't Anaconda for RaspberryPi so, for that reason, the library must be compiled from sources, this is one of the most problematic part of the project because the library has a lot of flags for compilation and after the compilation, since we use the library with python we need to create also the python binding and setup it correctly.

Since also this part is very technical, there is a section called **Install OpenCv3 with Python3** on the *robot setup* readme of the GitHub repository.

6.2.6 Samba

Since during the developing phase to share the code among pc and robot is important, I have setuped a shared folder using samba[7], in this way little modifications are faster to test.

Also for this setup there is the section called **Samba shared folder** on the readme of the robot setup part.

6.3 Hardware

In this chapter which hardware has been used to develop the prototype will be explained.

[7] https://www.samba.org/

6.3.1 WebCam

The webcam is one of the most important component of the robot. It allows to see what the robot sees.

The webcam is a Logitech c270, it has a maximum frame rate of 30 at 640x480.

The full camera specification is available at http://support.logitech.com/en_us/article/17556.

To use the maximum frame rate, all the camera functions useful to improve the image quality has been disabled.

Figure 6.1: Robot's webcam

6.3.2 RaspberryPi 3 Model B

The robot core consists of a RaspberryPi Model B, this is a embedded computer with arm architecture, in particular it has a Quad Core 1.2GHz CPU at 64bits, 1 GB of RAM, 4 USB 2 ports and a built-in wireless LAN.

The full specifications are available here https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

I chose this device because it is cheap but it can run an operating system like Debian, and it has also all the hardware capabilities that I need: the wireless LAN and the USB ports.

With more budget the hardware proposed by NVIDIA called Jetson TX2 (full specification here https://developer.nvidia.com/embedded/buy/jetson-tx2) should be very interesting. This one has an integrated GPU and CUDA capabilities which could allow to run all the tasks on robot instead of using a PC for heavy calculations.

Figure 6.2: Robot's brain

6.3.3 BrickPi3

Since my goal is to build the software for the robot I searched to have a simple hardware to avoid as much as possible electronic issues. For this reason I started to look at the lego mindstorm components (official website https://www.lego.com/it-it/mindstorms). Mindstorm provides lots of sensors and motors, but to connect them we need a "Brick" which is the "brain" of the LEGO robots.

As for our goal LEGO Brick is not powerful enough I make some researches and I found that a company called **Dexter Industries** developed a board which allows the raspberry pi to connect with LEGO's motors and sensors. This board is the BrickPi3[8].

With this board Dexter Industries create also a python library to control the LEGO components, mentioned at section 6.2.4.

6.3.4 LEGO Motors

The motion capabilities of the robot are possible thanks to the two **LEGO's Large Servo Motor EV3** [9].

[8]https://www.dexterindustries.com/brickpi/
[9]https://shop.lego.com/en-GB/EV3-Large-Servo-Motor-45502

Figure 6.3: Robot's motor controller

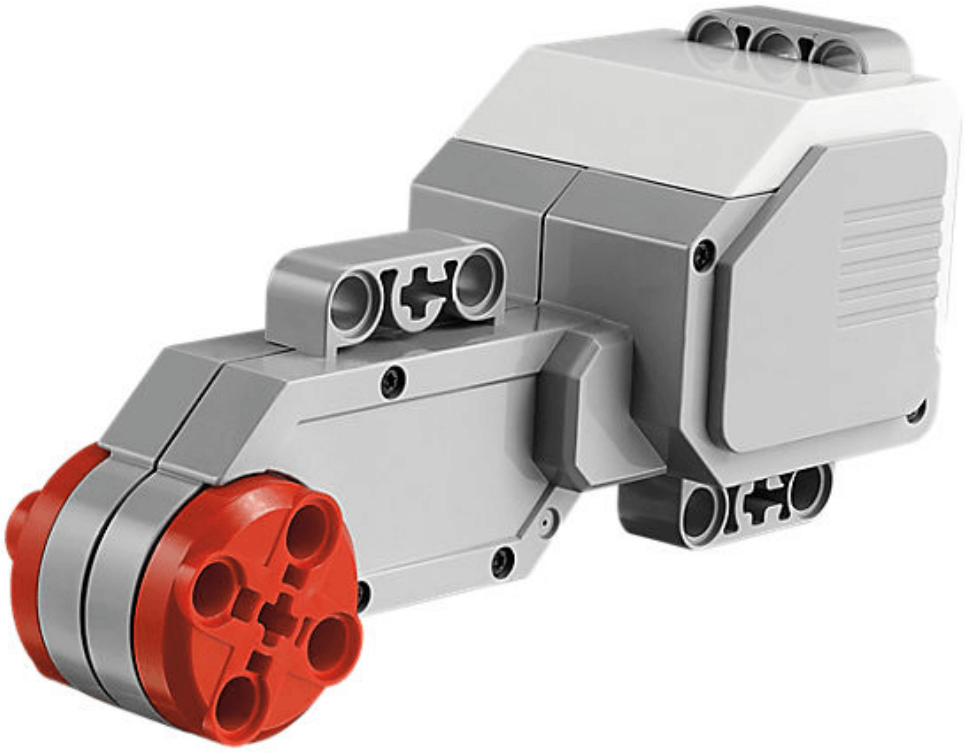


Figure 6.4: Robot's motor controller

6.3.5 Power supply

Since the robot uses lot of energy and motors, during the start it causes a voltage drop which involves a reboot of the raspberry, an additional power supply has been added. This power supply is a RPi PowerPack which has a battery of 3800mAh, in this way the raspberry pi doesn't reboot also if motors use lots of power.

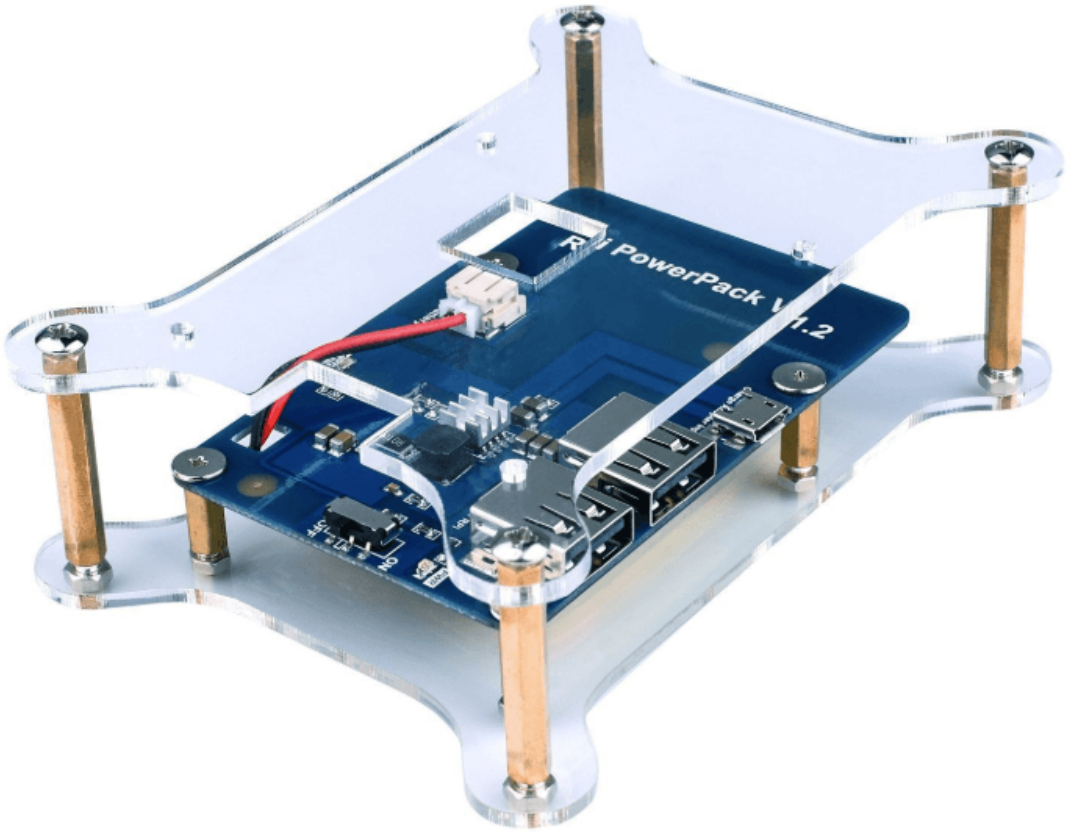


Figure 6.5: Robot's additional power supply

Chapter 7

Experimental results

This chapter reports the results and the performances of the algorithms developed and used in this thesis. These results should be considered as a starting point for future works and they show pros and cons of the choices that we did.

7.1 Performance evaluation of Advanced Lane detector

In order to be able to have a quantitative result of the quality of the **Advanced lane detector** that has been proposed, some metrics are required. We refer to the metric proposed in [33], in particular we considered the *Precision of Lane Feature Extraction (PLF)*, since in our approach we cannot calculate the accuracy (accuracy that has been used on the paper), and the *Lane Position Deviation (LPD)*.

In our approach the lane features are the median points, we do not have the *true negatives* and the *false negatives* so the accuracy cannot be calculated, on the contrary precision has been calculated as in 7.1, where TP are the *true positive* and FP are the *false positive*.

$$prec = \frac{TP}{TP + FP} \tag{7.1}$$

In addition to the above metrics, the number of un-detections has been considered too.

For our algorithm there is not annotated dataset with the appropriate format, for this reason such dataset has been manually generated from the KITTI [14] picking images at random according to the algorithms assumptions. In the dataset there are all the possible line features such as very sharp turn, difficult lighting condition and dotted lanes. During the annotation process the dotted line has been

connected in order to use the LPD metric.

The dataset composition is of 20 images divided into 4 categories as in 7.1, the sharp turns are the ones that are considered hard to detect since they are very sharp and the lighting conditions on them are difficult and full of shadows. Each image contains 2 lanes and for each original image two ground truth images have been generated, one for each lane.

strict lanes 1 side dotted	strict continuous lanes	not sharp turns	sharp turns
7	6	2	5

Table 7.1: Dataset structure

An example of an image with relative left and right ground truths is visible in figures 7.1,7.2 and 7.3

Figure 7.1: Dataset image

Figure 7.2: Left ground truth

The performance results obtained on such dataset have been reported in table 7.2

As we can see from the table 7.2, the main problem of this algorithm is related to the sliding window greedy approach that if there is a lot of noise, it will take the wrong decision and the line will be discarded during the residual quality checking.

Figure 7.3: Right ground truth

PLF	LPD (pixels)	missing detections %	fps
0.93	13.43	24%	12.4

Table 7.2: Performance metrics

The good news is that when the algorithm is able to detect the line it is very precise, so if in future works we are able to improve the filtering phases the algorithm performance related to the un-detection could be improved a lot.

In figure 7.4 is reported an example of evaluation, where the green points are the extracted lane features, the red line is the fitted line, and the white one is the ground truth.

For the first approach the performance was not evaluated because it has no sense since it is not a candidate for robot and also because it was difficult to adapt the format for such approach and the PLF could not be applied since there is not feature extraction phase.

7.2 Object Detectors results and performances

First of all it is important to notice that the results reported below could be improved, since how *Training* section 3.4.2 explains it was not possible to adjust hyper-parameters and execute the k-means algorithm on our datasets.

The detection performances has been evaluated using the metrics proposed in [10], where the IoU threshold has been set fixed to 0.5, if the IoU is below this threshold the object will be classified as undetected.

For detection phase the threshold has been setted lower for both networks this value has been chosen looking at the F1 Score. Precision and recall metrics are explained in 7.2.

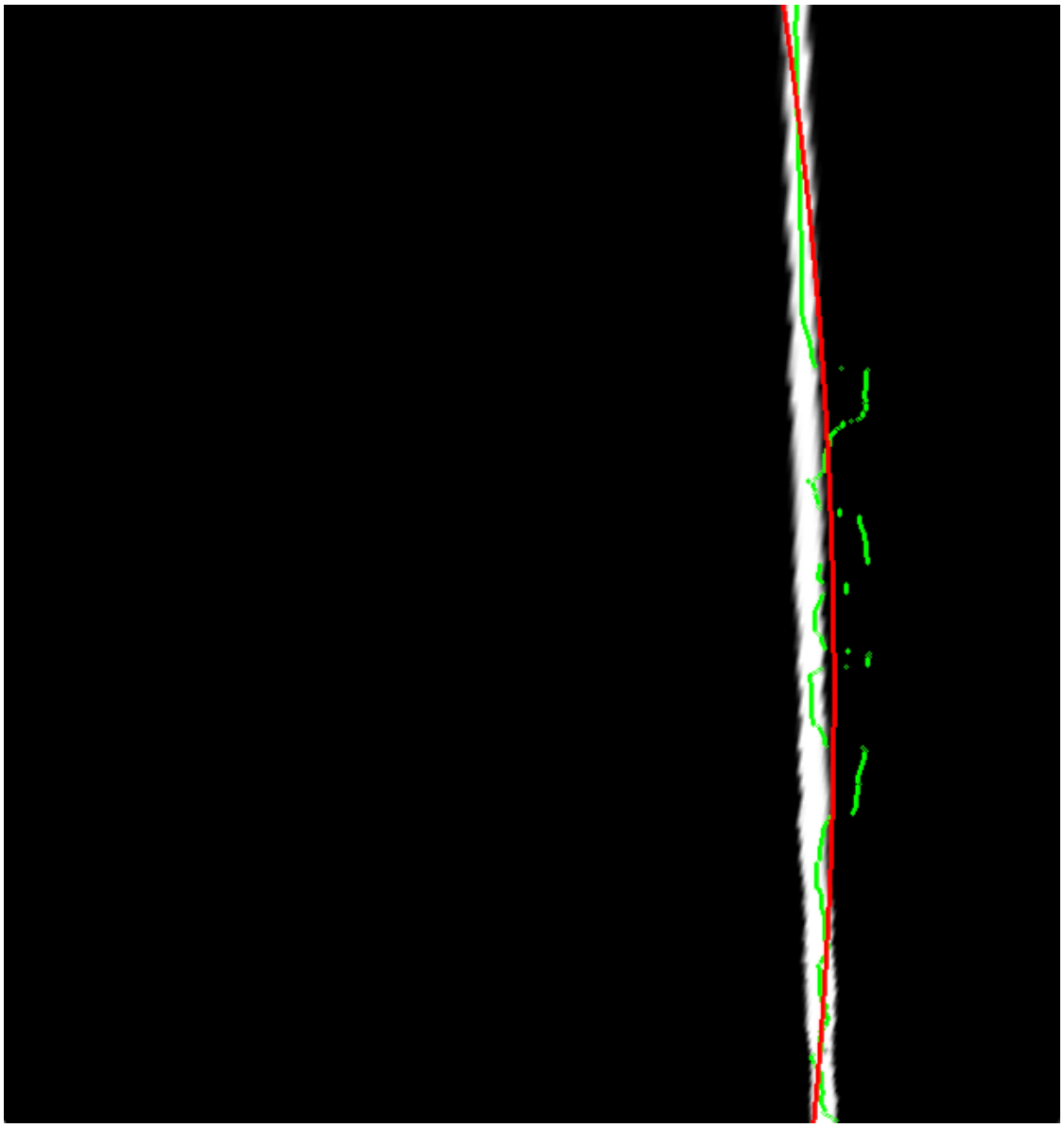


Figure 7.4: Visual representation of performance evaluation

$$prec = 2\frac{precision * recall}{precision + recall} \tag{7.2}$$

7.2.1 KITTI evaluation

The KITTI dataset has 9 classes but only "car", "person" and "Cyclist" are considered for the KITTI challenge [14] so also in this project only these classes have been considered.

The performances archived by the network on KITTI are reported in table 7.3

As we can see from the table, the detection performance is good for the "car" which is the very important in our project and poor for "person" and "Cyclist".

mAP	car AP	person AP	cyclist AP	fps (GTX 650 Ti)	fps (GTX 1070)
17.0	35.1	6.4	9.0	24.3	109.3

Table 7.3: Performance measures for KITTI

This fact is a big problem in real environment because people identification is fundamental for a self-driving car.

The first observation is that these classes have a different number of ground truth the class frequencies in the training set is reported in table 7.4, from the table it is easy to see that the class related to cars has lots of examples and the network is able to generalize well the features of a car, the other two classes instead have few examples. Another observation which comes from the table is that in general the bounding box size of these classes are smaller compared to the car's bounding boxes, and we know that YOLO has difficulties with small objects, even if YOLOv3 should be more precise it remains difficult to detect small objects for this architecture.

	Cars	Person	Cyclist
frequencies	20063	3240	1176
bbox area (px^2)	11501	6257	6945

Table 7.4: KITTI training set annotation frequencies and average bounding boxes area

On the official YOLOv3 paper [30], the tiny YOLOv3 trained on COCO [23] archives a mAP of 33.1, our configuration on KITTI reaches only 17.0 mAP, but as explained above the problem is related to difficulties on "person" and "Cyclist" detection. For "car" we are able to reach the 35.1 AP, and an important difference to highlight is that the number of annotations on COCO are more than 1.5 million of object instances. The larger number of instances which are present on COCO ensure better feature learning with relative higher mAP respect to KITTI.

In figures 7.5, 7.6 and 7.7 we can see the precision-recall curve of the respective classes "car", "person", "cyclist". In the figures we can see that class "car" has a good and stable curve, so the network has been able to learn how to detect them well. The other curves instead are more unstable, which means that the network was not able to extract well these classes, this instability is related to the few examples in the dataset too.

Considering the "car" curve, it is not so good because there is a strong decreasing pattern on it along recall axis. Having a decreasing pattern it is natural but a good detector should have a curve which stays at high precision, even if the recall grows up.

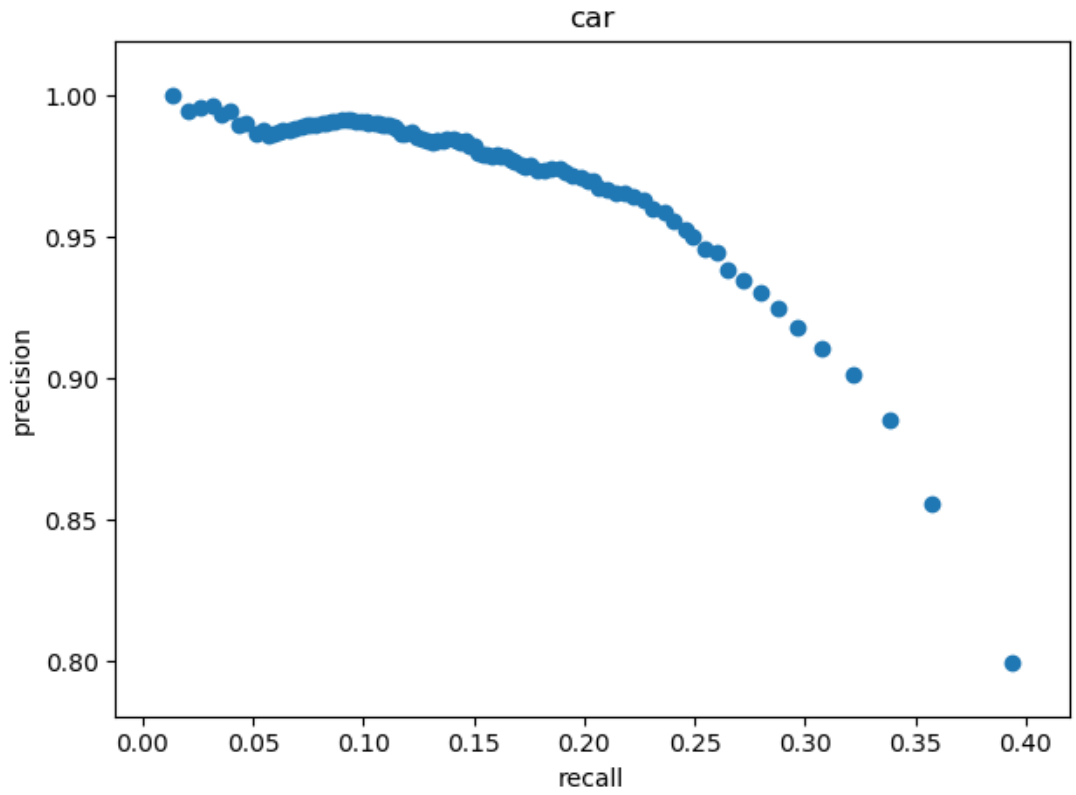


Figure 7.5: Precision-recall curve for "car"

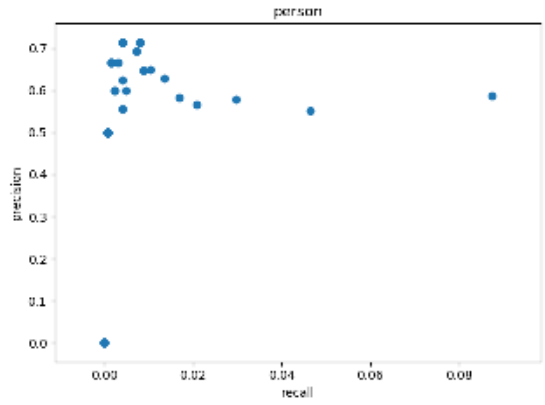


Figure 7.6: Precision-recall curve for "person"

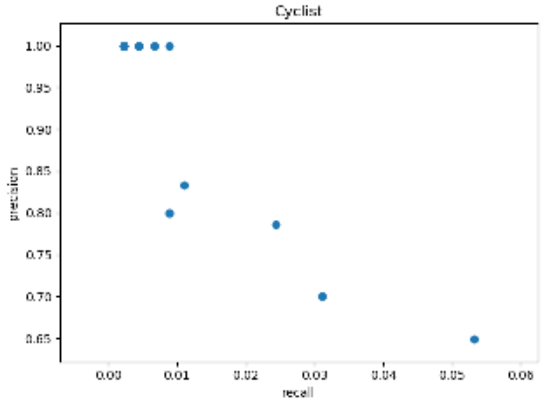


Figure 7.7: Precision-recall curve for "cyclist"

For what concerns the fps metrics, we are very happy since we are able to process images with a very high frame rate and we satisfy the real-time constraint.

Precision, recall and F1 Score have been evaluated for each class varying detection threshold from 0.01 to 1.0. The plots in figures 7.8, 7.9 and 7.10 show the results for KITTI, it is important to notice how the precision for "persons" and "cyclists" disappear respectively at threshold 0.4 and 0.2, instead for "car" it grows up: this

is related to the fact that "cars" class has lots of annotations and the network has learned how to extract better and with higher confidence compared to "persons" and "cyclists" such class.

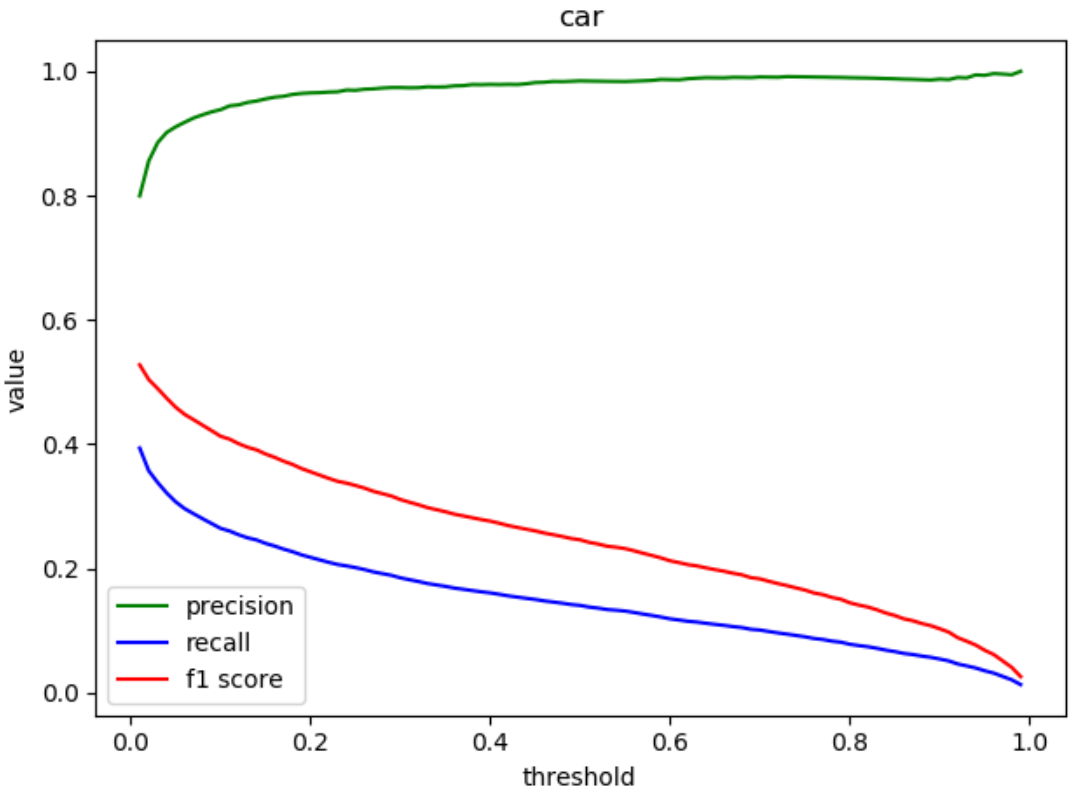


Figure 7.8: Precision, recall and F1 Score for "car"

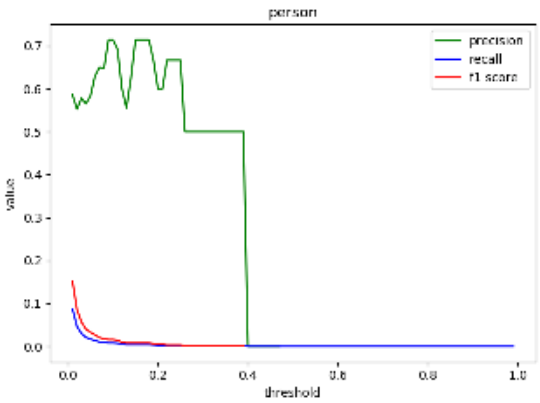


Figure 7.9: Precision, recall and F1 Score for "person"

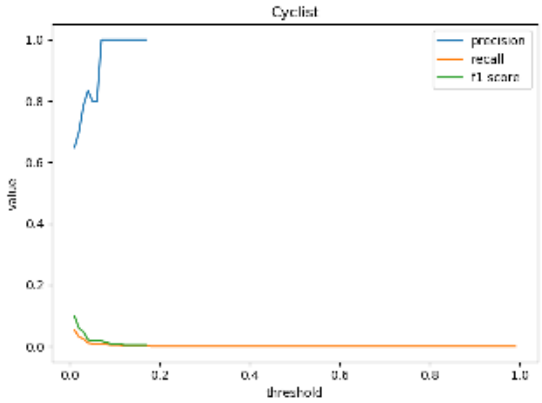


Figure 7.10: Precision, recall and F1 Score for "cyclist"

The main thing to notice is that the f1 score decreases and it is not good. This decreasing behaviour is related to the recall, it is not so high and it goes down

rapidly with the increasing of the detection threshold. Therefore this means that network has difficulties in detecting all the instances of an object in a given image, since recall measures the fraction of relevant instances that have been retrieved over the total amount of relevant instances.

The low value on recall metric is related to the KITTI structure too. Such dataset has lots of difficult annotations. An example of a difficult image with relative network detection is reported in figure 7.11, where it is easy to see that the network is not able to detect all the object instances and it confuses two instances in a single one resulting in fewer detections respect to the ground truth.

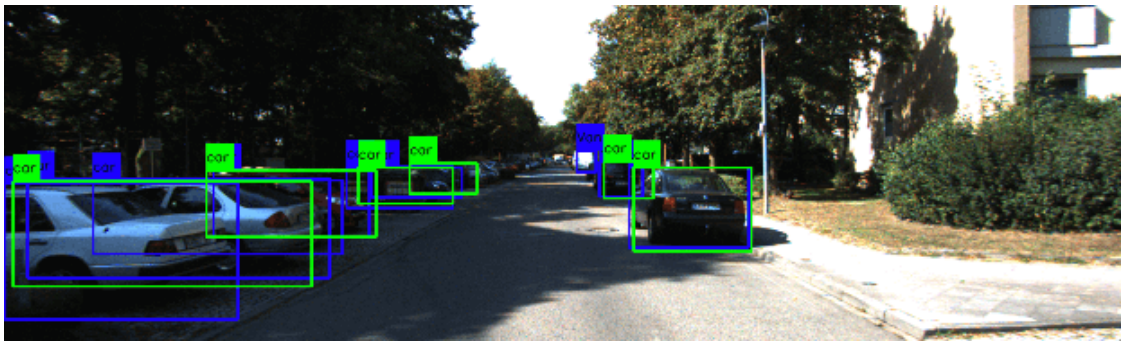


Figure 7.11: Difficult image with lots of cars

At a first impression seems that network does not work well, but this is not true. In our case the car approaches the various objects and so the size of such object becomes bigger when the robot approaches the object. In this situation the network is now able to detect the object, so in conclusion this network is fine for our goals.

There is an issue related to the difference among robot environment and KITTI dataset, the resolutions are different, in particular the issue is related to the *aspect ratio*: KITTI images resolution is 1242 x 375 so the aspect ration is 414 : 125, robot's view resolution is 640 x 480 with an aspect ratio of 4 : 3, so the KITTI images are wider and this has a big impact on the scaling performed by the network during detection phase. In figure 7.12, 7.13, 7.14 and 7.15 we can see how the same car will be transformed than in different input image resolutions (and aspect ratio) to the network's input.

From the images it is easy to understand that the network trained on the first transformation learns different bounding boxes compared to the second one, so to improve the detection on robot environment we need a new camera with an aspect ratio similar to the one in KITTI dataset. See also section 8 for further details.

Figure 7.12: KITTI's original image

Figure 7.13: KITTI's image resized for network

Figure 7.14: KITTI's image cropped to 640x480 resolution

Figure 7.15: KITTI's image cropped to 640x480 resolution resized for network

7.2.2 LISA evaluation

In LISA there are 47 classes, we considered only a subset because they were too many analyse. This subset consists on the set of classes that have an annotation frequencies higher than 100 on training set. The performances archived by the network on LISA has been reported in 7.5, the classes frequencies and their average bounding box area have been reported in 7.6.

mAP	pedestrian crossing AP	speed limit 35 AP	stop AP	signal ahead AP	fps GTX 650 Ti	fps GTX 1070	fps Intel Core i7 2,7 GHz
40.2	48.0	0.27	39.1	46.7	24.3	109.3	2.3

Table 7.5: Performance measures for LISA

	pedestrian crossing	speed limit 35	stop	signal ahead
frequencies	176	144	768	274
bbox area (px^2)	1679	915	1089	1953

Table 7.6: LISA training set annotation frequencies and average bounding boxes area

In the figures 7.16, 7.17, 7.18 and 7.19 have been reported the precision-recall curves for the analysed classes of the LISA dataset. The first observation is that the plots contains some patterns this is related to the fact that LISA dataset is composed by a sequence of images kept from different videos. The second observation is that these points are more "sparse" in comparison to KITTI, and this is caused by the fact that LISA has fewer annotations in comparison to KITTI, this is true in particular compared to "car" class.

The average bounding box areas between classes is similar and in general smaller compared to KITTI's ones.

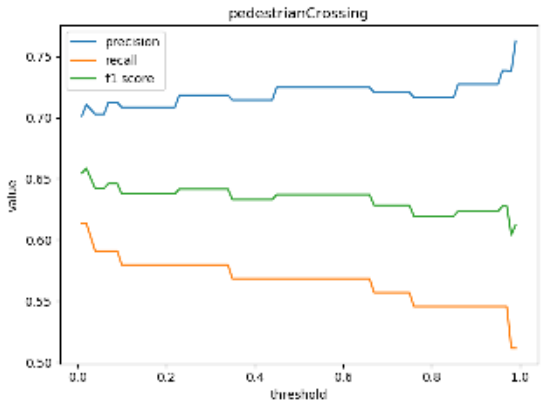


Figure 7.16: Precision-recall curve for "pedestrianCrossing"

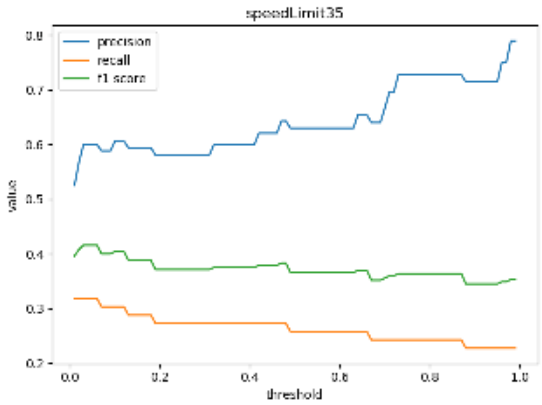


Figure 7.17: Precision-recall curve for "speedLimit35"

The precision, recall and f1 score plots are visible in figures 7.20, 7.21, 7.22 and 7.23. For the LISA dataset the situation is quite different compared to KITTI here the precision and recall remains more or less stable with different thresholds, the detected better sign is the "stop", following by the "pedestringCrossing" traffic sign.

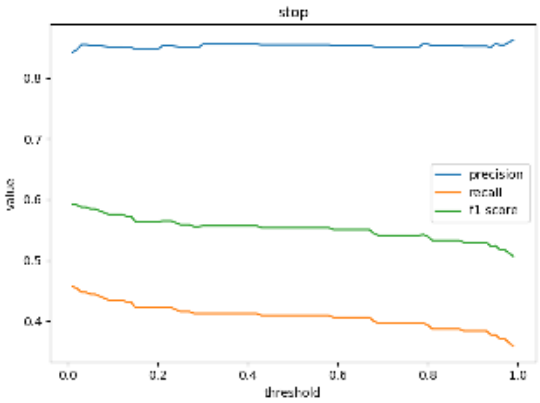


Figure 7.18: Precision-recall curve for "stop"

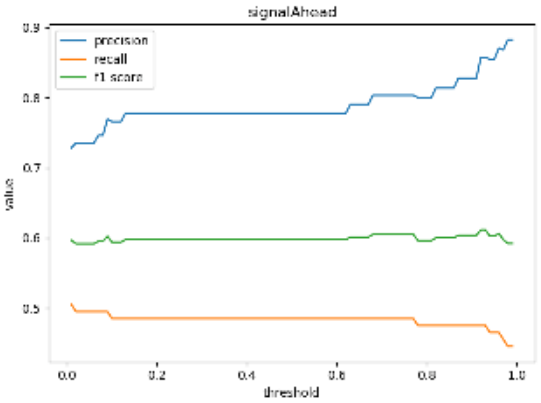


Figure 7.19: Precision-recall curve for "signalAHead"

LISA's "stop" class is detected very well, this is reinforced by the fact that the precision remains at high level even if the recall increases, as we can observe in figure 7.18.

The main difference between the network trained on LISA and the one trained on KITTI is related to the recall. In the network trained on LISA this value is higher than for KITTI, which means that the network is able to detect the major part of the relevant instances with a higher confidence, considered that both precision and recall are stable along the threshold axis. Also for this network the f1 score slightly decreases, which means that recall decreases faster that what the precision increases.

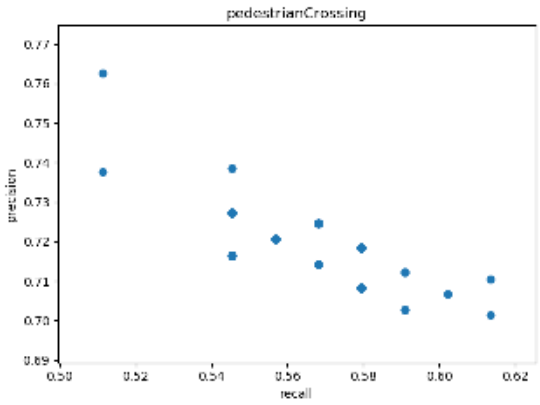


Figure 7.20: Precision,recall and F1 Score for "pedestrianCrossing"

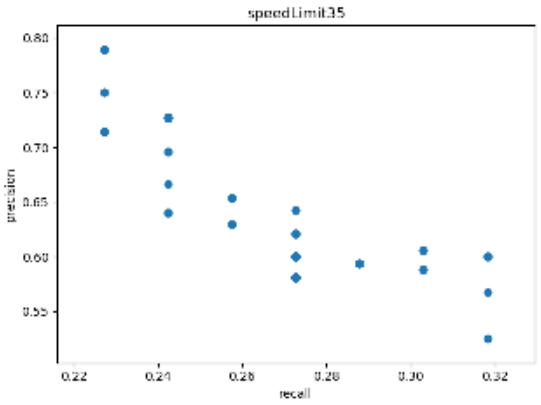


Figure 7.21: Precision,recall and F1 Score for "speedLimit35"

The LISA dataset has also the advantage of having blurred frames in its contents, these blurred frames are presented often in real driving environment, this characteristic helps the robot to detect the traffic signs while moving quite well.

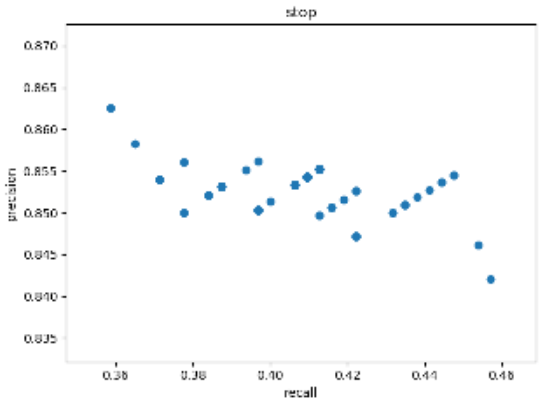


Figure 7.22: Precision,recall and F1 Score for "stop"

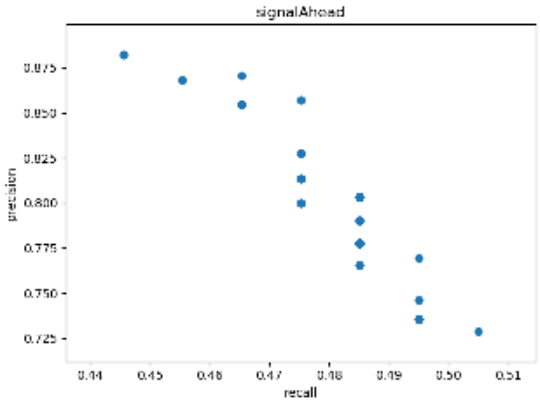


Figure 7.23: Precision,recall and F1 Score for "signalAhead"

7.3 Object Detectors comparison

Quite unexpectedly the network working on LISA archives better results compared to the one trained on KITTI. This situation is strange since KITTI provides more labelled images with a better quality compared to LISA. The difference is related to the object that networks must recognize.

The objects recognized by LISA are very simple: the traffic signs have always the same shape and representation what changes are the orientation and lighting conditions so if the network is able to learn such perspective transformations and it is able to extract specific feature from the image it is able to detect well this kind of objects; on the other hand KITTI's objects are very complex, cars, persons and cyclists have lots of possible positions and representations. Each person is different from the other and this means that the network has to learn more complex features to detect such objects and these features have not been learnt so well has we can see on the table 7.3. This situation affects the performances on robot environment too. Indeed, robot sees better the traffic signs compared to cars.

Chapter 8

Conclusion

In this thesis the engineering and computer science concepts have been merged to create a simple and working self-driving robot prototype.

The final result has reached the goal, we have a robot which is able to understand the surrounding environment and detect the typical objects that we meet during driving, like cars, pedestrians and cyclists. The robot is able to understand driving information and constrictions imposed by the traffic signs too. All these topics have been solved in real-time on a CPU computer interfaced with the robot equipped thanks to a very simple and cheap monocular camera as perception sensor.

The lane detector is very effective and reliable in the simulated environment, but for real-world application a deep learning approach for lane and road marking detection should be recommended [22].

The same considerations must be done for steering and throttling, where the naive version implemented in this thesis is very effective in the simulated environment but it should be replaced with a more complex steering logic by using end to end learning approaches, as proposed in [3].

The approach used for object detection has shown very interesting results and could be applied on both simulated and real environment. The suggestion to archive a higher precision and reliability of the detections for the real-environment, where cars have the possibility to have a powerful computer installed on it, is to use the original YOLOv3 [30] version and not the tiny one. This should be particularly useful for the network related to KITTI, since the tiny version it is not so precise on car detection in the real world.

A fundamental issue to be considered while using "deep learning" or in general supervised learning approaches is the dataset characteristics. These characteristics like weather conditions, lights and camera, used to capture images, affect a lot what the network is able to learn. These features represent the most important issue to

keep in mind and the dataset must be chosen or generated with precision and it must be strongly focused on what is the goal the network wants to archive. Each situation that our network must be understand needs to be present in the dataset with a sufficient number of examples given at the network the ability to learn each possible situation. In KITTI for example all the images have been taken in San Francisco in very sunny days so the network responsible to detects cars, pedestrians and cyclists works only under very sunny weather conditions.

Merging engineering topics with theoretical results has been very challenging and often it shows unexpected results, since metrics evaluated on datasets are, often, very dataset dependant and if our real environment has different characteristics compared to the dataset algorithms should perform worst than expected.

This project can be the starting point for lots of future works, As explained in 8.2.

8.1 Issues

Several issues have been met during the project development; lots of them have been solved and is how this has been done explained in the relative sections. Here we highlight the unsolved and the partially solved issues.

Having two networks instead of one it is problematic and a single network in future building should be fixed. The problem of having a single one is that we need a dataset which contains object classes for the two network and such dataset does not exist at the time of writing.

Another problem related to object detection is that the YOLO architecture has problems on small objects detection. This problem is mitigated by the fact that the YOLOv3 [30] introduces multiple scales detection. And it is not particularly important in self-driving cars, since the object that has to be detected is the one which is closer to the car, so in general the big one.

Others issues are related to the lane detector. This approach is very sensitive to noise, in roads with shadows the threshold includes also areas in sunshine and these areas may effect the maximum on the histograms.

This lane detector could be improved, for example the parallelism assumptions has not used yet, their introduction could improve a lot the robustness of the detector and if also the information of the previous frames take into account the algorithm, we may have another improvement.

To recap the lighting condition the more relevant issue remains and it affects the results of both lane detection and object recognition.

Another issue, easy to be solved, is the following: in the final implementation the computer, which has been used to communicate with the robot, is a MacBook Pro that has not CUDA capable graphic card, so the whole system is running on such configuration. With this configuration the object detection runs at 2 frames per second so the distance threshold for car detection has been removed during experiments but it is easy to reintroduce it when the algorithm runs on CUDA capable GPU.

Another issue is related to the camera used to capture frames, this camera has a resolution of 640x480 with an aspect ratio of 4:3 it has a limited view capabilities and for lane detector is too zoomed and it is not able to detect lines closer to robot and for object detections it has a limited width range. The first problem has been solved by putting the camera in a higher position with an angle of 45 degrees respect to the road plane, in this configuration the lanes could be well detected. The second one is not a real problem, but only a limitation. Finally the last one could be overcome using a different camera with a higher width range, dedicated for the object detection tasks.

8.2 Future works and improvements

In conclusion this project is very interesting and suggests opportunities for future works. In particular it should be interesting adding a navigation module. An interesting paper that should be used as starting point is [6] which is based on navigation through a monocular camera.

Further work should be done to improve the lane detector. First of all the techniques that allows to consider information of previous frames to reinforce the lane detection and remove better noises could be implemented. Examples of these approaches can be found in [1] and [5].

Furthermore robust techniques could be used instead of lane detector, but more robust deep learning techniques requires a hardware upgrade.

Such hardware upgrade should be the introduction of a different robot's core as the NVIDIA Jatson TX2, which has CUDA capabilities. With this device all the tasks should be moved on robot, archiving a more robust and interesting robot. This device is quite expensive and it is for this reason that has not been used in this project, since all the hardware used has been bought by the thesis's author.

Bibliography

[1] Mohamed Aly. "Real time Detection of Lane Markers in Urban Streets". In: *The IEEE International Conference on Intelligent Vehicles Symposium (IV)*. 2008.

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Ed. by Springer. 2006.

[3] Mariusz Bojarski et al. "End to End Learning for Self-Driving Cars". In: *CoRR* (2016).

[4] Amol Borkar and Monson Hayes. "A Novel Lane Detection System With Efficient Ground Truth Generation". In: *IEEE Transactions on Intelligent Transportation Systems* 13 (2011), pp. 365–374.

[5] Amol Borkar, Monson Hayes, and Mark T. Smith. "Robust lane detection and tracking with ransac and Kalman filter". In: *The IEEE International Conference on Image Processing (ICIP)*. 2009.

[6] Jakob Engel, Thomas Schöps, and Daniel Cremers. "LSD-SLAM: Large-Scale Direct Monocular SLAM". In: *European Conference on Computer Vision (ECCV)*. 2014.

[7] Martin Ester et al. "A Density-Based Algorithm for Discovering Clusters". In: *The Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*. 1996.

[8] M. Everingham et al. *The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results*. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.h

[9] M. Everingham et al. *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results*. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.h

[10] Mark Everingham et al. "The Pascal Visual Object Classes (VOC) Challenge". In: *International Journal of Computer Vision* 88 (2010), 303—338.

[11] Open Source Robotics Foundation. *Robot Operating System (ROS)*. URL: http://www.ros.org/ (visited on 03/14/2018).

[12] Andrei Furda and Ljubo Vlacic. "An Object-Oriented Design of a World Model for Autonomous City Vehicles". In: *The IEEE International Conference on Intelligent Vehicles Symposium (IV)*. 2010.

[13] Andrei Furda and Ljubo Vlacic. "Enabling Safe Autonomous Driving in Real-World City Traffic Using Multiple Criteria Decision Making". In: *IEEE Intelligent Transportation Systems Magazine* 1 (2011), pp. 4–17.

[14] Andreas Geiger et al. "Vision meets Robotics: The KITTI Dataset". In: *International Journal of Robotics Research (IJRR)* (2013).

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Ed. by The MIT Press. 2016.

[16] Google. *Tensorflow Object Detector API*. URL: https://github.com/tensorflow/models/tree/master/research/object_detection/ (visited on 03/14/2018).

[17] Google. *Tensorflow website*. URL: https://www.tensorflow.org/ (visited on 03/14/2018).

[18] Zicheng Guo and Richard W.Hall. "Parallel thinning with Two-Subiteration Algorithms". In: *ACM* 32 (1989), pp. 359–373.

[19] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* (2017).

[20] Jonathan Huang et al. "Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017.

[21] Dexter Industries. *Dexter Industries website*. URL: https://www.dexterindustries.com/ (visited on 03/14/2018).

[22] Seokju Lee et al. "VPGNet: Vanishing Point Guided Network for Lane and Road Marking Detection and Recognition". In: *The IEEE International Conference on Computer Vision (ICCV)*. 2017.

[23] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *CoRR* (2014).

[24] Wei Liu1 et al. "SSD: Single Shot MultiBox Detector". In: *European Conference on Computer Vision (ECCV)*. 2016.

[25] Andreas Mogelmose, Mohan M. Trivedi, and Thomas B. Moeslund. "Vision based Traffic Sign Detection and Analysis for Intelligent Driver Assistance Systems: Perspectives and Survey". In: (2012).

[26] OpenCv. *OpenCv 3.3 website*. URL: https://opencv.org/opencv-3-3.html (visited on 03/14/2018).

[27] Joseph Redmon and Ali Farhadi. *Darknet*. URL: https://pjreddie.com/darknet/ (visited on 03/14/2018).

[28] Joseph Redmon and Ali Farhadi. *YOLO: Real-Time Object Detection*. URL: https://pjreddie.com/darknet/yolo/ (visited on 03/14/2018).

[29] Joseph Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017.

[30] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *arXiv* (2018).

[31] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[32] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2016), pp. 1137–1149.

[33] Ravi Kumar Satzoda and Mohan M. Trivedi. "On Performance Evaluation Metrics for Lane Estimation". In: *The IEEE International Conference on Pattern Recognition (ICPR)*. 2014.

[34] Marco Signoretto. *drAIver GitHub repository*. URL: https://github.com/MarcoSignoretto/drAIver (visited on 03/14/2018).

[35] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Ed. by Springer. 2010.

[36] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks 5th edition*. Ed. by Pearson Education. 2011.

[37] Trieu. *Darkflow*. URL: https://github.com/thtrieu/darkflow (visited on 04/04/2018).

[38] Satoshi Tsutsui, Tommi Kerola, and Shunta Saito. "Distantly Supervised Road Segmentation". In: *The IEEE International Conference on Computer Vision (ICCV) Workshops*. 2017.

[39] M. N. A. Wahab, N. Sivadev, and K. Sundaraj. "Target distance estimation using monocular vision system for mobile robot". In: *The IEEE Conference on Open Systems (ICOS)*. 2011.

[40] Zhengyin Zhou et al. “End to End Learning for Self-Driving Cars”. In: *International Journal of Digital Content Technology and its Applications* 5 (2011), pp. 192–202.