POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Politecnico
di Torino

Master's Degree Thesis

Environment perception for an autonomous radio-controlled vehicle with artificial intelligence algorithm

Supervisors

Prof. Nicola AMATI

Prof. Andrea TONOLI

Eng. Stefano FERACO

Eng. Sara LUCIANI

Candidate

Andrea BERETTONI

Academic Year 2020 - 2021

Abstract

In the last years Autonomous Vehicles have become one of the most important and popular automotive topics. A concept that was considered futuristic a few decades ago is ready to enter our lives and completely change the experience of driving and the whole transportation. Many research studies predict a huge positive impact related to driving aspects such as comfort, low traffic, and safety. In general, an autonomous vehicle's control consists mainly of three separate modules: environment perception, planning and decision-making, and vehicle control. Environment perception is defined as the process of interpreting vision and sounds. It is a process to interpret, acquire, select, and then organise the sensory information from the physical world to make actions like humans. Therefore, among the technical problems that self-driving vehicles have to address, perception is one of the most challenging.

This thesis work was done in collaboration with the student Team Squadra Corse Driverless. According to the race rules, the racetrack boundaries are formed by cones of two colours: blue for the left line and yellow for the right line. For this purpose, the vehicle's control is not done in a conventional way as lane keeping or similar, but in such a way that vehicle should perceive whether a cone is present and which colour has. In this context, the project work focuses on creating a rapid platform for the PoliTo SC19 electric race-car using a 1/10 radio-controlled vehicle able to test the perception algorithms identified during this project. They are the Single Shot Detector (SSD) and You Only Look Once version 4 (Yolov4) for object detection allowing real-time cone detection. As a first step, a preliminary research on object detection techniques is carried out analysing all the possible Convolutional Neural Networks. Then, the final hardware configuration of the RC car is done wiring all the sensors used for this project that are: stereo camera, Nvidia Jetson Xavier and IMU. In parallel, considerable work is addressed with the software setups using the Robot Operating Systems (ROS) environment. Here, the perception pipeline node performs the object recognition employing either the SSD or the YOLOv4, i.e. it extrapolates the data coming from the stereo camera and gives back the bounding boxes of the cones. Finally, the results and discussion are presented both in a simulation environment and in experimental tests performed in Aeroclub Torino.

Acknowledgements

Al termine di questo bellissimo e tortuoso viaggio vorrei, innanzitutto, ringraziare il Professor Amati ed il Professor Tonoli, per la fiducia riposta in me dall'inizio di questa avventura. Vorrei, inoltre, dire grazie a Stefano e Sara per avermi seguito e guidato in questi mesi con la Vostra immensa pazienza e disponibilità.
Un profondo ringraziamento a tutti i ragazzi incontrati all'interno del Laboratorio Interdisciplinare di Mecattronica con cui ho condiviso ogni giorno degli ultimi 7 mesi, Eugenio, Gennaro, Marco, Raffaele, Salvatore e Stefano. Mi avete dato tanto!
Ringrazio la divisione di Perception & Planning di Squadra Corse Driverless, ed in particolar modo il team leader Sebastian, per aver creato un ambiente dove il piacere di lavorare insieme ci ha permesso di raggiungere risultati importanti.

Vorrei ringraziare ora tutte le persone che mi hanno permesso di arrivare a tagliare questo traguardo. Ringrazio la mia famiglia e i loro sacrifici, reputati erroneamente scontati e dovuti, per avermi dato la possibilità di inseguire i miei sogni. Grazie dal profondo del cuore!
Un grazie anche a tutti gli amici e colleghi incrociati in questi due anni, in particolare a Mattia e Filippo, siete stati degli ottimi compagni di viaggio.
Ai ragazzi del Club, siete sempre con me.
Il mio ultimo ringraziamento va a Benedetta, a cui devo la forza e il coraggio di questi anni. Abbiamo incontrato molte difficioltà ma ancora una volta ci siamo riusciti, insieme. A te dedico ogni secondo di questa vittoria.

Nel prepararmi ad affrontare le prossime sfide che mi si prospetteranno davanti, di nuovo, un sentito Grazie a tutti Voi.

"To infinity and beyond"
Andrea

Table of Contents

List of Tables

List of Figures

Acronyms

AI
Artificial Intelligence

FSD
Formula Student Driverless

CNN
Convolutional Neural Networks

AV
Autonomous Vehicles

ADAS
Advanced Driver Assistance Systems

SSD
Single Shot Detector

YOLO
You Only Look Once

ADS
Automated Driving Systems

ANN
Artificial Neural Network

mAP
mean Average Precisions

FPS

Frame Per Seconds

LiDAR

Light Detection And Ranging

SLAM

Simultaneous Localization and Mapping

ROS

Robot Operating Systems

EKF

Extended Kalman Filter

UKF

Unscented Kalman Filter

GRV

Gaussian Random Variable

RC

Radio-Controlled

SAE

Society of Automotive Engineers

MSE

Mean Square Error

Chapter 1

Introduction

In the last years the concept of autonomous driving has become one of the most important and famous in the automotive field. Academic and automotive industries are involved in substantial research activities moving toward the construction of a fully autonomous car aiming to change how people perceive transportation completely. This chapter aims to introduce the thesis topics, starting from a brief introduction of the Autonomous Vehicle (AVs) history, the worldwide opinion and the description of all the autonomous driving levels. A brief description of Formula Student's competition is provided, describing all the different classes but focusing on the driverless section. Finally, the thesis motivation and the thesis outline are presented, paying particular attention to the different stages done in this project. This thesis work is done together with Squadra Corse Driverless (SCD). It is a team working on converting an already existing prototype car – the SC19 of the SquadraCorse PoliTo – into a full autonomous vehicle according to the Formula Student regulations, under the supervisor of the LIM laboratory at Politecnico di Torino.

1.1 Background

Vision of AVs in the first decades of the 20^{th} century remained a futuristic concept that only the science fiction enthusiasts trying to imagine. For example, in 1958, one of the companies that changed the world like Disney, aired a program titled "Magic Highway USA" that imagined a future in which AVs were guided by coloured high-way lanes. Only a few years later, in the mid-1980s the underlying computing and other technologies needed to realise this beautiful dream start to become available. Three-time periods can be highlighted to understand the advances and milestones made in the following years [1].

Phase 1: Fundamental Research

From approximately 1980 to 2003, a main research period devoted to the basic studies of autonomous transportation was done in partnership between university research centers and some transportation and automotive companies. Two main technology concepts emerged from this work. Firstly, researchers pursued the development of automated highway systems, in which they analyzed how vehicle depends significantly on the high-way infrastructure. Therefore, to corroborate their thesis some demonstrations were done, however one of the major demonstrations took place on 1997, over a 7.6-mile stretch of California's 1-15 highway near San Diego. It was done by the California Partners for Advanced Transit and Highways (PATH) program, where the "DEMO 97" program shown the platooning of eight AVs guided by magnets embedded in the highway and coordinated with vehicle-to-vehicle (V2V) communication [2]. The second idea was to develop semi-autonomous and autonomous vehicles that are not dependent on highway infrastructure. In 1980, a team led by E. Dickmanns at Bundeswehr University Munich in Germany developed a vehicle with vision-guide technique, able to navigate at speeds of 100 kilometres per hour without traffic. Moreover, at Carnegie Mellon University's NavLab was developed a series of vehicles from NavLab 1 to NavLab 11, from the mid-1980s to 2000s. An important role was taken by NavLab 5 in the 1995, where during a tour called "No Hands Across America", steered autonomously for 98 percent of the time while a human operator controlled the throttle and brakes.

Phase 2: Grand Challenges

From 2003 to 2007, the U.S. Defense Advanced Research Projects Agency (DARPA) held three "Grand Challenges" that considerably increased AVs technology development [3][4]. The first two of them challenges the research world teams in the developing of a fully autonomous vehicle that might participate in a 150-mile off-road race with about a million rewards. Although no vehicle was able to complete the 2004 Grand Challenge, in the 2005 Grand Challenge course five teams reached the end of the race, where the fastest team complete the race in about seven hours. The third challenge was done in 2007, titled the "Urban Challenge". As the name suggests, vehicles raced through a 60-mile urban course, navigating with other autonomous and human driving vehicles. Six team was able to complete the course. This challenges lead to a huge increasing of the technological components as sensor systems and computing algorithms used to detect and react at the environmental conditions.

Phase 3: Commercial Development

Due to the DARPA challenges, many partnerships between car makers and academic sector arise, increasing the AVs knowledge much faster. In particular, companies playing a crucial role in the autonomous field are Tesla[5], Google[6] and nuTonomy[7]. Research today estimates that owning a smart autonomous vehicle will become the normality in 2050. Although the esteem price of a self-driving car is still outside the price range of most consumers, investor interest continues to increase together with the decreasing for the manufacturing costs as well as the improvement of the technology used. According to a research done in the University of Texas, whether 90 percent of the car present on the United States roads are replaced by autonomous vehicles, the savings across various industries, such as automakers, insurers and the government, might reach $450 billion.

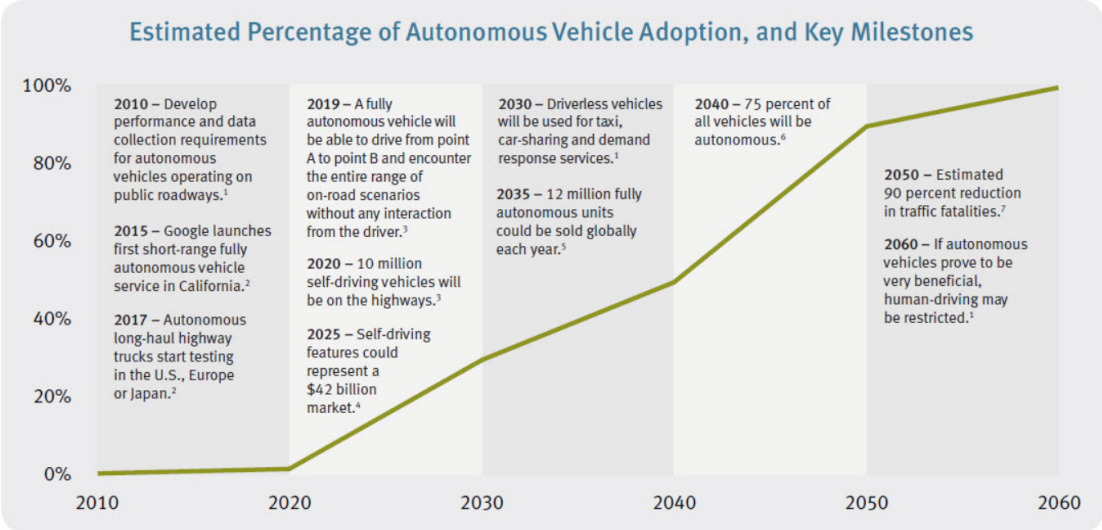


Figure 1.1: Estimated Percentage of Autonomous Vehicle Adoption

Nowadays, AVs are able to arouse great interest worldwide, deploying a huge research effort to address design challenges related to performances and safety involving broad categories starting from the traditional automotive OEMs and a significant number of start-ups and newcomers. In the last years the development of Advanced Driver Assistance Systems (ADAS) and self-driving is one of the most important automotive topics [8]. The ADAS are groups of electronic technologies that assist drivers in driving and parking functions. Through a safe human-machine interface, it increases car and road safety using automated technology, such as sensors and cameras, to detect nearby obstacles or driver errors, and respond accordingly. The use of these components in the vehicle is one of the first steps

toward the complete autonomous car, they are already present in most of the vehicles of the recent mass productions, mainly thanks to the development of robust safety systems and new technologies able to provide a real-time assessment of vehicle's dynamics and passengers' comfort. However, fully autonomous vehicles are not ready to be introduced in the automotive scenario, there are multiple causes that lead to this conclusion, but one of the most important is the technological limitations of the solutions that are available at that moment. The research efforts are oriented on affordable safety-oriented solutions and in the reliability of the available systems. Moreover, every year a not negligible number of people are involved in road accidents. Unfortunately, even if objectively modern vehicles are safer than those produced in the past, the number of people killed worldwide continues to be high for vehicle passengers and for external road users such as pedestrians and cyclists. In the majority of the cases, these accidents are due to the driver's fault. Therefore, they can be theoretically replaced by self-propelled cars.

Autonomous Vehicles are about to change the way we perceive and we live transportation. People's needs and demands are changing fast together with the development of new innovations, this lead to a consistent improvement of life's quality in terms of environmental impact and safety. In fact, automated vehicles are considered to be efficient, with a consistent fuel reduction with respect to the human driving and sustainable for safe driving. Obviously, the autonomous driving will change the driving experience itself. Although the first prediction says that a huge positive impact in terms of traffic flow, comfort, allocation of time resource and safety will be guaranteed, the main problem is to convince people of the advantages and ensure safety and comfort for the passengers. When driver or passenger has no direct influence or a small control of the car, the car should behave and generate movements that are perceived as acceptable from the human point of view. Hence, a relevant aspect from the car maker is to analyse accurately the vehicle motion's effects on human body to make a prediction on the feelings of people during the trip [9].

1.2 SAE Driving Autonomous Level

Talking about AVs from a more technological point of view, to understand the different automated technologies, in 2014 the Society of Automotive Engineers (SAE international) provides through its SAE J3016 "Level of Automated Driving" [10] a taxonomy with detailed definitions of six levels of driving automation spanning from no driving automation (level 0) to full driving automation (level 6). This guide has been updated in 2018 and is considered as the reference for autonomous driving technologies.

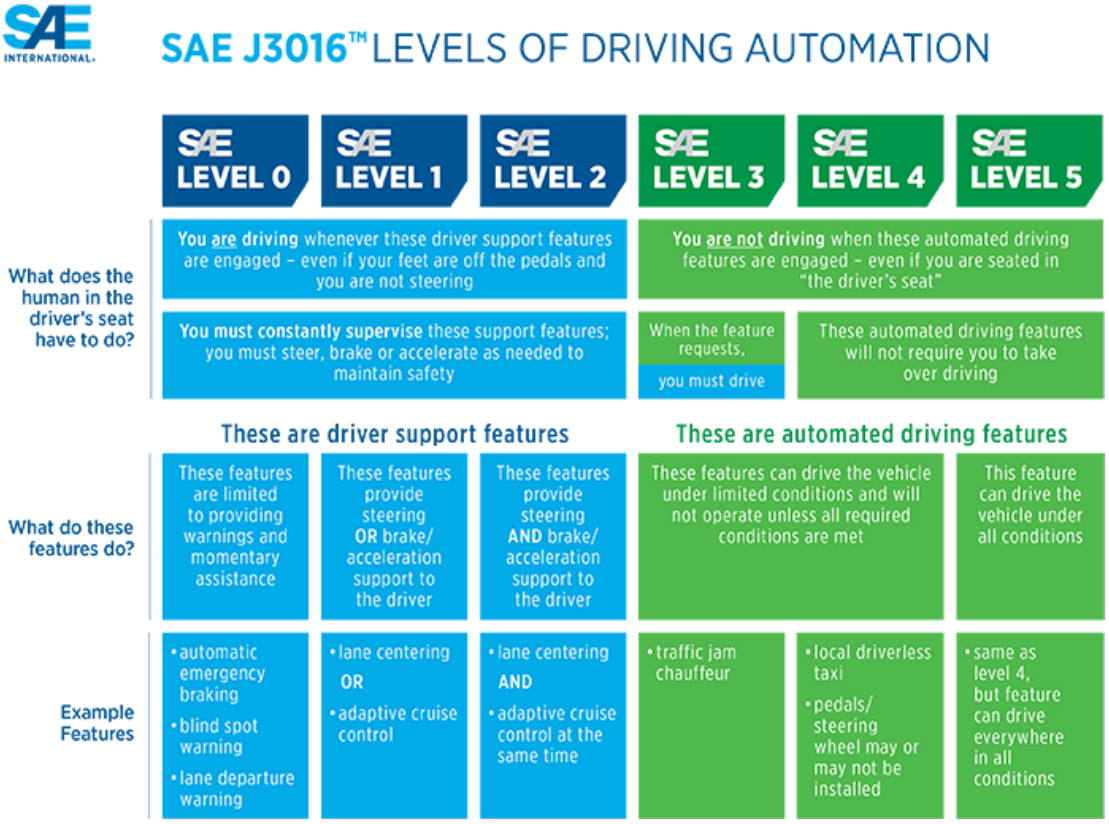


Figure 1.2: SAE level of Driving Automation

The classification is based on the amount of driver intervention and attentiveness required rather than the vehicle capabilities. The most automated transportation systems that is already available on the market achieve the Level 2 of automation, in which the driver should monitor the system and decide when take the lead of the vehicle. However, Level 3 and Level 4 are already achieved and tested in different scenarios where the conditions are similar to the external world. On the opposite, the technology used in that moment does not guarantee the achievement of full automation (Level 5). This is due to the environment in which the AV is operating and the conditions which a human driver is naturally able to consider. However, SAE's levels are devoted on technical rather than legal aspects. Thus, they clarify the role of the Automated Driving Systems (ADS) which are progressively included in the vehicles. With the term ADS is considered both hardware and software tools able to perform dynamic driving tasks.

The six levels of classification are defined as follow:

- **Level 0**: *No driving automation.* All the dynamic tasks (DDT) are completely performed by driver even though the vehicle is equipped with active safety

systems. The systems under this level are found in conventional vehicles.

- **Level 1**: *Driver assistance*. The vehicle's System is capable to perform one of the manoeuvres – lateral or longitudinal – but not both simultaneously. Here, under a specific conditions a given automation system shares the control of the vehicle with the driver.

- **Level 2**: *Partial Driving Automation*. The vehicle's System is capable to perform both lateral and longitudinal manoeuvres under the complete supervision of the driver that must be prepared to intervene immediately at any time. A Level 2 vehicles are equipped with a wider set of ADAS.

- **Level 3**: *Conditional Automation*. The vehicle becomes capable of taking full control under well-defined driving scenarios, but the driver must be always in the condition of suddenly taking back control when required by the system. The driver's full attention is required during the action.

- **Level 4**: *High Driving Automation*. Human attention and interaction are not needed anymore. The vehicle's System is capable to perform the full driving task over limited scenarios and environment conditions. Self-driving is supported only in limited spatial spaces and for this reason pedals and steering wheel are still present in the vehicle to guarantee to drive in scenarios that go beyond the defined uses cases.

- **Level 5**: *Full Driving Automation*. The vehicle's System will have capabilities to function in all the scenarios and there won't be any contingency safety system needed in case of critical situations. From now on, only the concept of passenger exists.

1.3 Formula Student Driverless Competition

Formula SAE is an international university engineering design competition initially proposed by the Society of Automotive Engineers (SAE) which involves the design and production of a racing car, evaluated during a series of tests based on its design qualities and engineering efficiency. Started in the USA in the 80s, Formula SAE challenges involved students' team from all around the word involved to design, build, test and race a small-scale formula-style race car. Until 2016, the main categories were: Class 1C (combustion engine vehicles), Class 1E (electric vehicles). Then, some months later, a third category was added by Formula Student Germany (FSG) called 1D, devoted for self-driving vehicles. In fact, in parallel with the huge research interest in autonomous driving and technological aspects they presented the first ever European FSD event, the Formula Student Germany Driverless 2017.

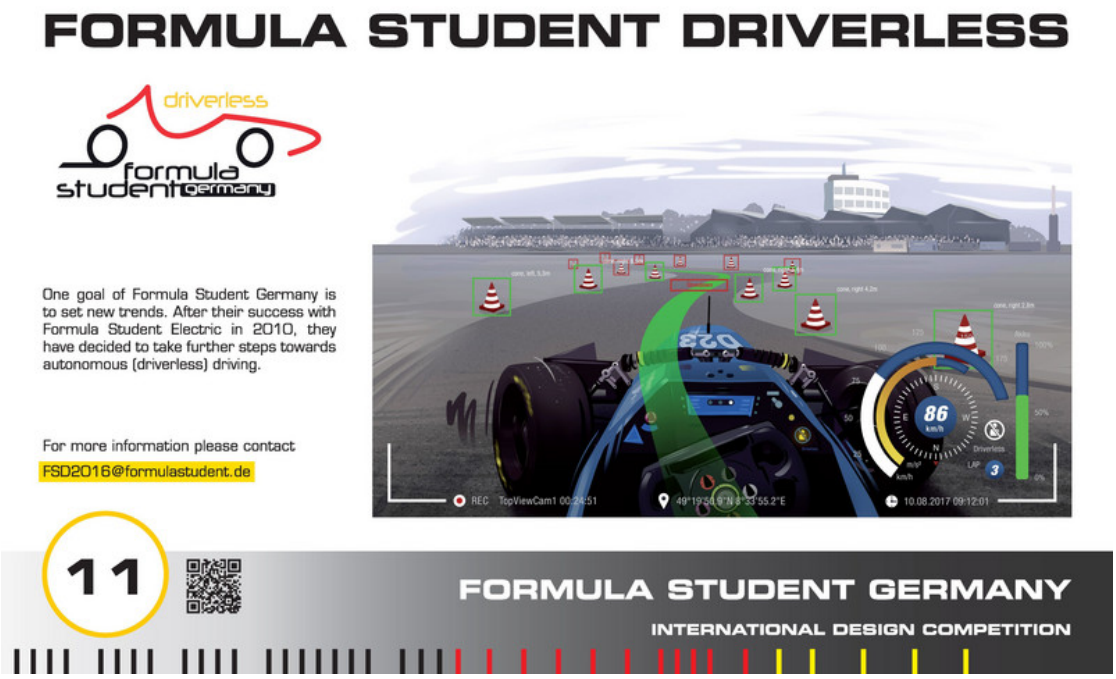


Figure 1.3: First presentation of Driverless event in Formula Student Germany 2017

Since the main objective was to develop a race car that can run without a driver in Autonomous mode, the vehicles must fulfil the technical requirements related to the type of feeding and the self-driving classes. Each students' team create a prototype based on a series of rule [11], whose purpose is to guarantee on-track safety; indeed, to minimize any risks, the autonomous competition is done in a secured, person-free test area. Although might seems that who would win the competition should be the fastest on track, it is not completely right. In fact, as in the other existing competition classes, the combines static and dynamic events are what counts for the final victory.

The FSD challenge is to make in practise all the theoretical background that students learned during the university period, making a practical experience in building and manufacturing the car as well as considering management aspect that are proper of the automotive industry. This part is belonging to the static discipline where a jury coming from the motor-sport world evaluate the prototype at engine off and the sales plan that might match given criteria as engineering innovation, construction quality and cost planning. Then, the dynamic discipline is fully done on the track where the students should demonstrate how the self-built car behave in the unknown environment. Indeed, the most challenging part is that

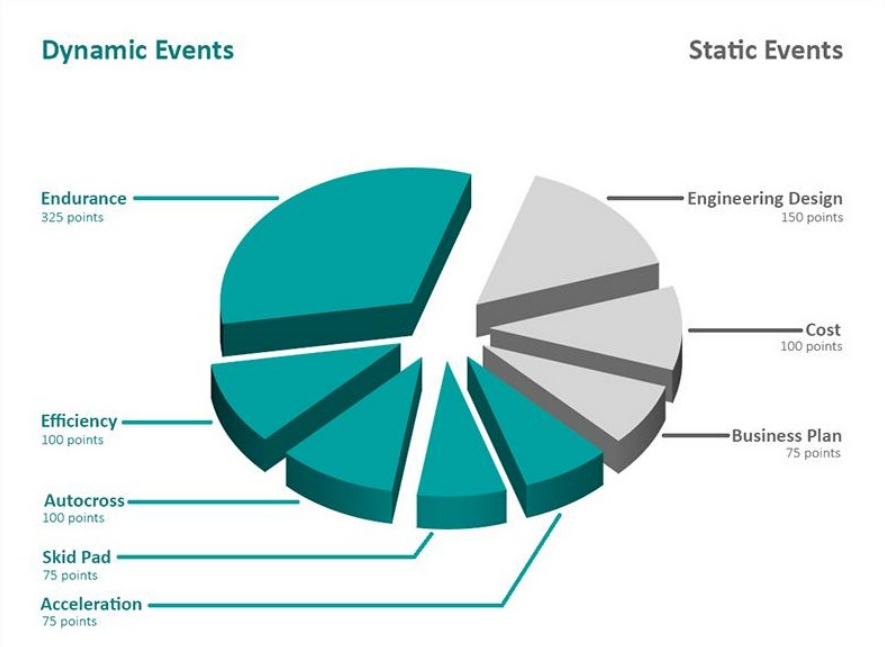


Figure 1.4: Score of the Formula Student Driverless competition

the layout of the track is not known a priori. Therefore, in the first lap, the vehicle might be able to move inside the boundary that are delimited by cones through the use of the information coming from a stereo camera and LiDAR. Thanks to these informations, the trajectory planning strategy is to calculate the path that the vehicle has to follow avoiding dynamic instability failure of the system problems. After the vehicle completes the first lap, the circuit map is saved in such a way that the car from the second lap on already know the track and trajectory to follow adopting an aggressive driving set aiming to minimize the lap time. The vehicle must run along a track that is delimited by cones for 5 km distance – about 10 laps – as fast as possible without any human pilot intervention or remote-control systems. The cones have dimension 228 x 335 mm with three different classes of colour that are blue cones and yellow cones are used to delimited the left and right boundary respectively, while the orange ones delimited the starting point of the race.
The goal is to convert an AWD single-seater racecar into a self-driving vehicle competing in the world's largest racing competition, Formula SAE, driverless category starting from the SC19 prototype as shown in the figure 1.5. This vehicle was designed and developed by the student's Team Squadra Corse PoliTo. In the process of building a full autonomous system, it is possible to highlight three macro-areas that are Perception, Motion estimation and Mapping, and finally Control. In particular, this thesis work is mainly focusing on the perception pipeline to

allow cone detection using two different neural networks. The ADS is developed at Politecnico di Torino by the research group of Prof. Andrea Tonoli and Prof. Nicola Amati at DIMEAS – LIM Mechatronic Lab. In 2022, the team will present its first driverless prototype.

Figure 1.5: Squadra Corse PoliTo SC19 prototype

1.4 Thesis Motivation

The aim of this thesis project was to design and implement a rapid prototyping of the SC19 using an 1/10 radio-controlled vehicle. In particular, for this thesis purpose, two computer vision algorithms for object detection are considered involving in a specific scenario of Formula Student Driverless (FSD). Since the lanes in this competition are formed by cones, the traditional control approach as lane-detection is not applicable here. Therefore, it was crucial to analyse algorithms that focuses on particular specifics of the application such as high-speed driving and cone-based lane markings.

1.5 Thesis Outline

The Thesis is organized as follow:

- *Chapter 2*: it presents the basic knowledge of the Machine Learning and the Object Detection task together, with a partiuclar focus on the Convolutional Neural Networks chosen.
- *Chapter 3*: it presents the Hardware configuration explaining all the sensors configuration. Secondly, after a brief overview about the Robot Operating

System (ROS), the Software architecture is presented analysing the main topics of the autonomous pipeline.

- *Chapter 4*: It presents the perception algorithm description analysing how the two Convolutional Neural Networks are implemented in the ROS environment. Then, the rapid platform developed using the radio-controlled vehicle is shown to test the perception pipeline.
- *Chapter 5*: It presents the results coming from both the simulation environment and the real tests implemented in the Aeroclub Torino.
- *Chapter 6*: It is the final chapter, where conclusion and future works are reported.

Chapter 2

Machine Learning and Object detection

2.1 Introduction

The aim of this chapter is to introduce to the reader the Machine Learning (ML) concepts. This is a fundamental step aiming to present how the further techniques used in this thesis work. This chapter is divided in two main blocks, the first one contains an high level description about machine learning and deep learning, talking about some historical notes and how they work. The second part is devoted to explain the concepts behind ML and some basic mathematical models with the purpose to contextualize what is an artificial neural network and how it works.

2.2 Historical Notes about AI, ML, DL

Nowadays, the words "Artificial Intelligence" are used very often in a latitudinal of different scenarios. However, the meaning of the terms Artificial Intelligence (AI) and machine learning (ML) is confused and not so clear. Obviously, all of their concepts are related by the historical timeline linked with the technological improvement. AI was used for the first time in the 1950s; then, the term ML was used by A.Samuel in 1959 [12], while Deep Learning (DL) was born recently. The mutual relation between them can be described with concentric circles as shown in Figure 2.1. Hence, AI is the precursor of all the other concepts.

Looking for a formal definition, the Artificial Intelligence is the ability of a computer or a robot controlled by a computer to do tasks that are usually done by humans because they require human intelligence and discernment [13]. Differently, Machine Learning is an application (a sub-field) of AI that provides systems the

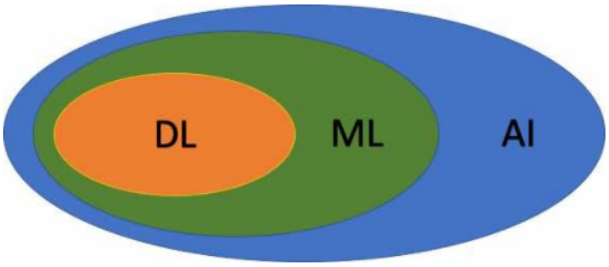


Figure 2.1: Relation between AI, ML, DL

ability to automatically learn and improve from experience without being explicitly programmed [14]. Finally, Deep Learning is a branch of ML characterized by the usage of deep artificial neural network that means learning models with great number of layers and neurons [15].

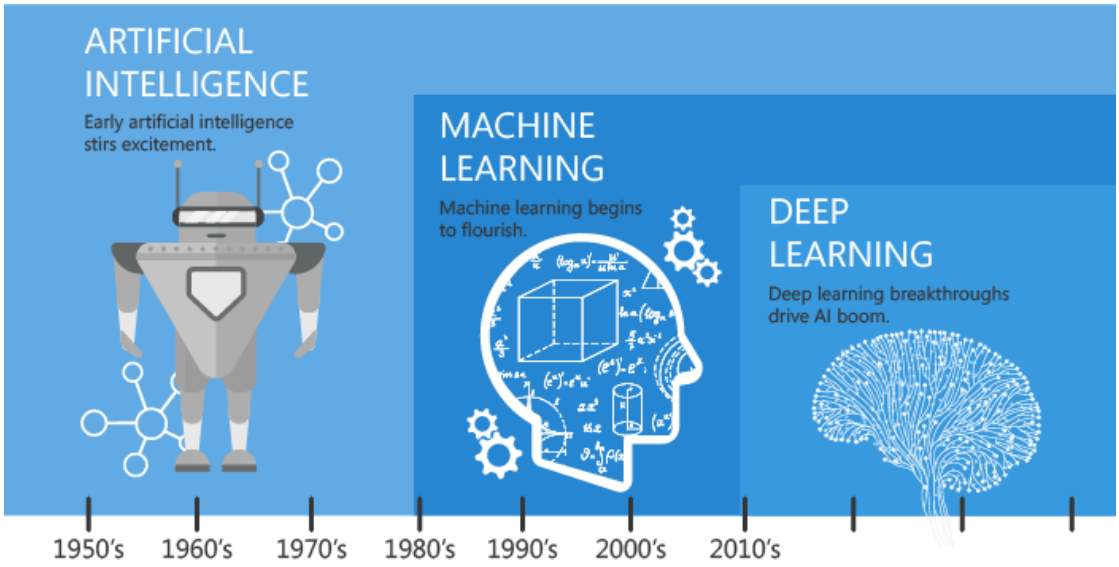


Figure 2.2: Artificial Intelligence, Machine Learning and Deep Learning. [16]

2.2.1 History of Deep Learning

The history of Deep Learning can be traced back to 1943, when Walter Pitts and Warren McCulloch created a computer model based on the neural networks of the human brain [17]. They modelled a neuron with a simple linear binary classifier looking to the sign of the function $f(x, w) = x_1 w_1 + \ldots + x_n w_n$ they were able to distinguish between two different classes of inputs. However, the weights were

chosen manually by the human. Moving forward, in the 1950s the Rosenblatt's perceptron comes. It was able to adjust the weights with a first idea of an iterative training process that it caused a great success. Another important contribution comes by Widrow and Hoff in 1960 with the Adaptive Linear Element (ADALINE) where it uses a first version of the stochastic gradient descent aiming to change the weights. It is one of the fundamental training algorithm since it is widely used also nowadays. However, linear models present limitations due to the small complexity, for example they have problems in learning the XOR function. All these problems was highlighted by Minsky and Papert in 1969 exposing some negative considerations. This brought to a loss of interest in the next period that was called the "first winter" of AI. In the 1980s a movement called *connectionism* recreated the bases for a new prolific period about neural network. It gives a wide contribution about Deep Learning with the concept called distributed representation where the inputs can be represented by many shared features. The framework for artificial neural networks was decided with the idea of a model for Parallel Distributed Processing (PDP), exploiting multiple connected neurons (units). Hence, the idea of the human brain was inspired firstly by Rumelhart and McClelland, and together with Williams they conceived the concept of hidden layers and the back-propagation algorithm, which is one of the actual widely used method to train models.

The second important period proceed till the end of the 1990s where Long short-term memory (LSTM) aiming to model long sequences of informations discovered by Hochreiter ans Schmidhuber. From 2000s on, a research report by META Group described the increasing volume of data and the increasing speed of data as increasing the range of data sources and types. This was a call to prepare for the onslaught of Big Data. So the combination between computation possibilities and the improvements in the model efficiency brought Deep Neural Network with a big consideration outperforming other ML models.

The third wave is still going on, where the main focus is devoted to small dataset or with unsupervised learning techniques (i.e. without the usage of labels for data). The progressively importance of the Deep Learning might be justified by some key factors. Firstly, in the "Big Data" epochs, huge datasets are available to train deep models thanks to the computational capability that increase day by day. A rule of thumb suggests that a dataset of 10 millions of elements is the starting point to reach the human ability to classify items of different classes. For example, very important for the Deep Learning era is the ImageNet dataset composed by 14 millions of images of about 20.000 different cathegories. In fact, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) organized around the dataset creation from 2010, had a strong impact. In 2012, through a convolutional neural network called AlexNet was possible to obtain a top-5 error of 15.3%, that was an amazing results with respect to the previous attempts. This result has been improved in the following years using models as VGG, GoogleNet and ResNet.

Another fundamental key factor has been the spreading of powerful computational hardware as fast CPUs but in particular general purpose GPUs. Nowadays, GPUs are built with such as an architecture able to provide optimized parallel operations, where in some cases, are properly devoted to the Machine Learning purposes. Making a comparison with the high number of neurons presented in the brain and in the artificial neural network, it is possible to see that while humans have about 10^{11} neurons, today, bid artificial neural networks reach about 10^6 neurons. Moreover, the number of connections per neuron was initially limited by the hardware capability.

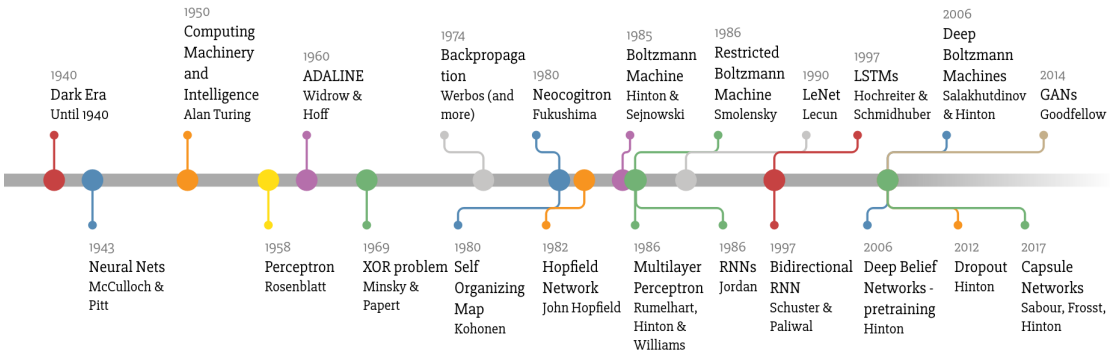


Figure 2.3: Timeline of Deep Learning History

2.3 Machine Learning Concepts

In this section a brief description of Machine Learning concepts is proposed. The goal is to provide to the reader a basic understanding of how neural network works and which are the main strategies and difficulties related to their implementation.

2.3.1 The artificial neuron: Threshold Logic Unit (TLU)

From the historical evolution of Deep Learning presented in the previous section, it is immediate to say that the concept of artificial neural networks has been inspired by the biological model of the human brain. Obviously, the modern evolution of Machine Learning mainly relies on statistics, numerical optimization and mathematics; however, neuroscience should be considered as a crucial source of inspiration. A schematic model of the biological neuron is reported in 2.4.

The main elements which compose a biological neuron are [19]:

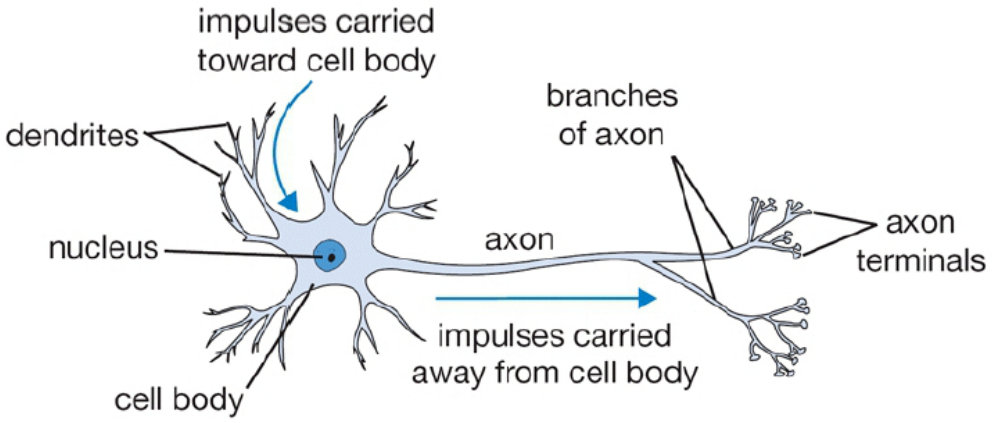


Figure 2.4: A schematic model of the Biological Neuron [18]

- Cell body or Soma: it encloses the nucleus of the nerve cell;
- Dendrites: they are branched extensions which allow the cell body to receive input signals from neighboring neurons;
- Axon: it is a particular and unique extension connecting the cell body to the synapses. It is responsible of carrying the electrical signal;
- Synapses: they are a ramified structure located at the final section of the axon. They pass information to the other neurons.

Roughly speaking, the concept behind neural activity is based on the flow of electrical signals from a neuron to the closed ones. More specifically, the signal flow is managed by electrochemical processes such as voltage-gated ion exchange to let the electrical signal move through all the neuron's cells. To justify this process, let's follow the entire path of the signal. If it passes through the axon termination, it will be transmitted to the synapses. Here, a certain amount of neurotransmitters is released. This substance has a significant influence on the synapse conductivity in such a way that it can attenuate or boost the signal; in some sense, it can be depicted as weights. Then, once passed the synaptic junction, the signal is forwarded to the post-synaptic neuron thanks to dendrites, which are able to capture neurotransmitters. Local small currents are created by the positive and negative signals arriving from the dendrites in the soma. Here, they can be mixed together and summed up. At the end, when the soma electrical potential reaches a certain threshold, an impulse is generated and transmitted again along the axon.

Now, by looking at a simple artificial neuron model (Figure 2.5) called the Threshold Logic Unit (TLU), a parallelism with the human neuron previously

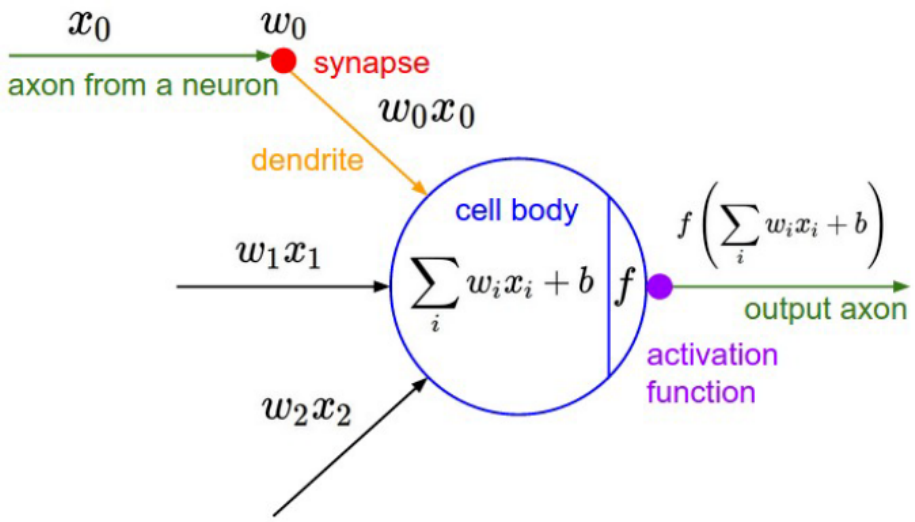


Figure 2.5: An Artificial neuron model with a generic activation function [18]

described will be explained. Firstly, starting back from the axon, a numerical signal between 0 and 1 becomes the input signal for another neuron through the synapses. At this point, a weight w_i is assigned related to the synapses' conductivity level. Then, all the weighted input signals from the dendrites are summed as it happens in the soma. Finally, the signal is propagated depending on which activation function is chosen. "Threshold" is the name of a basic activation function where the weights between the connections are not equally distributed but depend on their priority. They also are excited or inhibited according to the intensity of the chemical transmitter; in general, all these processes happen for artificial neural networks (ANNs) with weights and inhibitory signals.

Moreover, according to the first model of an artificial neuron, the following assumptions were written by the McCullogh-Pitts:

- The activation function of each neuron is an established threshold theta;
- The output is binary (logic unit);
- Input signals are identically weighted and can be inhibited;
- At each time step the output signal will be equal to 1 if the sum of all the weighted inputs is greater than the threshold and the neuron is not inhibited, 0 otherwise.

Summarizing, the behaviour of an artificial neural network can be represented

as follow:

$$output = \begin{cases} 1 : \sum_{i=1}^{n} w_i x_i \geq \theta \wedge \text{no inhibition} \\ 0 : \text{otherwise} \end{cases} \tag{2.1}$$

2.4 Architecture of Artificial Neural Networks

As already explained in the Deep Learning history in Subsection 2.2.1, Linear classifiers cannot deal with non-linear problems; for example, one of the most challenging problems in the past was learning the XOR function. To fix it, the evolutionary architecture of Artificial Neural Networks (ANNs) resulted in being successful. As shown in Figure 2.6, a generic structure is obtained by connecting the neurons with a precise organisation. Neurons vertically grouped formed what is called a layer. In particular, the first layer on the left is also named the input layer since its neurons are directly connected to input signals. In contrast, the last layer on the right is called the output layer responsible for labelling the numerical signals arriving from previous layers. Finally, in the middle, there are the hidden layers. The main elements of basic neural network architecture are:

- The input signals x_i;
- The weights w_{ij} and bias b_j of each connection;
- The activation function a_j of each neuron.

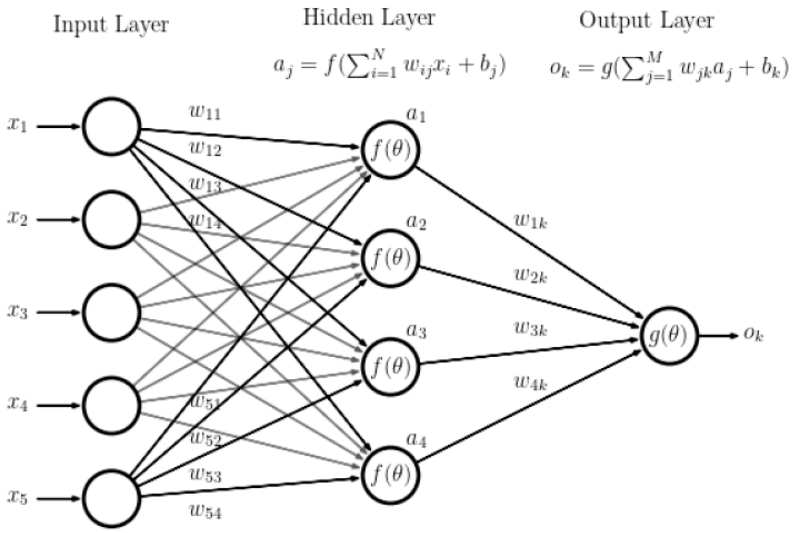


Figure 2.6: Example of Neural Network Architecture

The weights role is to decrease or boost the signal of each connection. The activation functions used in the hidden and the output layers are usually different, according to the specific function to learn. Once it arrives at the neuron, the activation function decides to switch the signal on or off. A general way to write the activation is:

$$a_j = \sigma\left(\sum_{i=1}^{N} w_{ij}x_i + b_j\right) \tag{2.2}$$

2.5 Activation Functions

The "Learning Process" of a neural network consists of the adaptation of its weights and biases. However, if an activation function with binary output is used, small parameter changes lead to a significant difference in the output. Then, a slight variation of the weights can cause a switch from 0 to 1 of the output and vice versa.

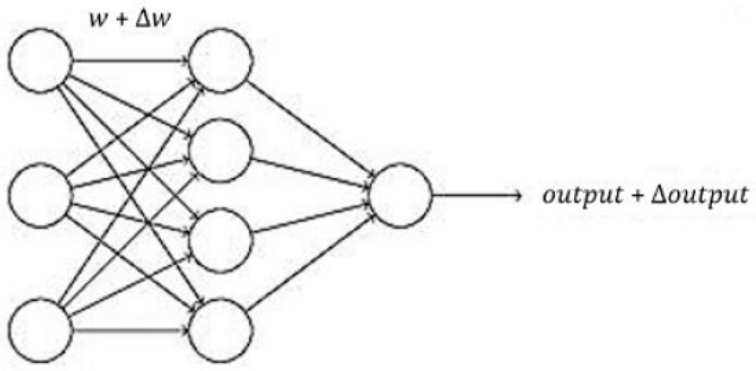


Figure 2.7: Propagation of the weight's variation till the output

Sigmoid Function

To overcome the problem depicted previously, the sigmoid function is used with the form:

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.3}$$

It provides a smoother variation of the output since it remains bound in a limited range. Today the sigmoid activation function is one of the most used in Machine Learning.

Linear Activation Function

A linear activation function is of the form:

$$\sigma(z) = cz \tag{2.4}$$

It produces an output that is proportional to the input. However, since the output is not binary, it is straightforward to understand that having a network with all neurons with linear units will be a linear signal. Moreover, since the output of a neuron will be weighted and then feed-forward to the next neuron, there is the possibility to replace multiple layers with just an equivalent one.

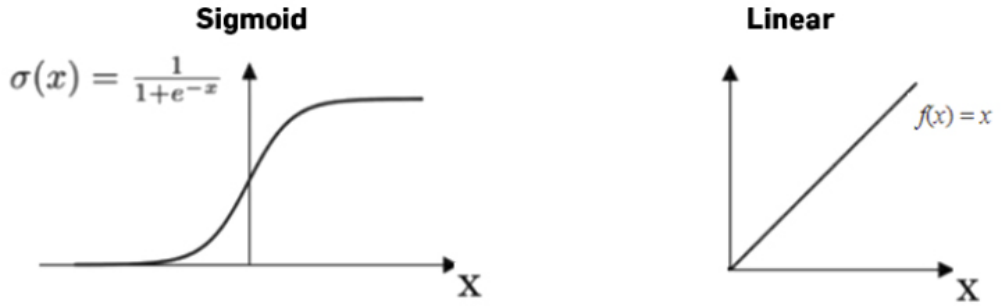


Figure 2.8: On the left the Sigmoid Activation function showing the smooth behavior between 0 and 1. On the right the Linear Activation function showing a canonical behavior of a ramp starting from the origin

Tanh Activation Function

The hyperbolic tangent activation function is similar to the sigmoid function. The main difference between the two depends on the range of the output values since $\tanh(z)$ presents an output that spans from -1 and 1. In general, it is less used than the Sigmoid function; however, it can give some benefits in a particular application. The expression of this function is:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.5}$$

While the expression for the activation function of the neurons considering $\tanh(wx+b)$ can be formulated with:

$$\tanh(z) = \frac{1 + \tanh\left(\frac{z}{2}\right)}{2} \tag{2.6}$$

Rectified Linear Unit (ReLU)

The ReLU function is defined in the following way:

$$\sigma(z) = \max(0, wx + b) \tag{2.7}$$

From the equation 2.7 it is evident that the shape of this activation function is a classic ramp for positive value, 0 otherwise. Although it is easy to guess that the function is similar to a linear unit activation function, ReLU has relevant advantages. Firstly, its non-linearity gives good approximation properties, and it can restrict the network from a computational point of view since it allows only a limited part of neurons to fire. It also involves simple mathematical operations with respect to sigmoid function. ReLU is probably the most used activation function in Deep Learning.

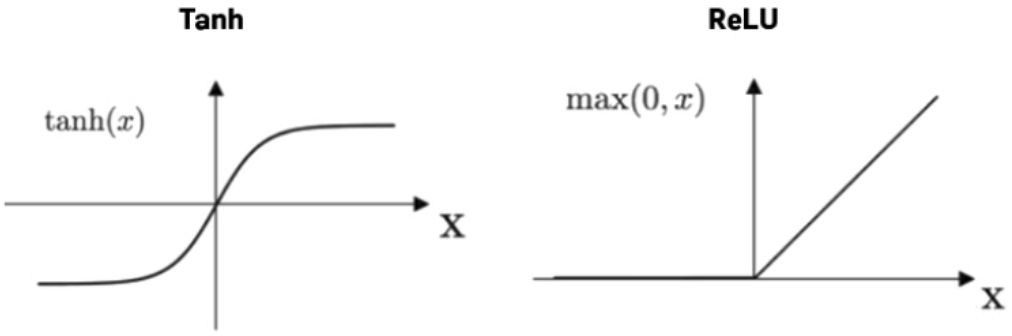


Figure 2.9: On the left the Tanh Activation function showing a smooth behavior between -1 and 1. On the right the ReLU Activation function showing the discontinuity around the origin

2.6 Learning Process: Gradient Descent

In the previous paragraphs, the architecture of a generic artificial neural network is presented. Now, the main concepts about the learning process of Artificial Neural Networks will be analysed. Firstly, the purpose is to get a collection of biases and weights for the model that, according to the assignment, gives us the correct output. So, it is helpful to have a way to measure and evaluate how the network is adapting to its weights. For this purpose, a cost function is introduced.

$$C(w, b) = \frac{1}{2n} \sum_{x} ||y(x) - a||^2 \tag{2.8}$$

In the expression 2.8, w and b are respectively the weights and bias of the network, n represents the total amount of samples used for the learning process. The value x is referred to a single training input and the desired output is written in the form y(x), while the actual output obtained from the network is named a. There are multiple ways to write a cost function; in this particular case, this quadratic cost is known as *Mean Square Error* (MSE). In general, it measures the error committed between the output that is predicted with the desired one. Of course, when $C(w, b)$ is around 0 for all the training inputs, the training algorithm is working properly and a good set for w and b has been found. Hence, a minimisation problem for C(w,b) has to be carried out.

As previously state, the number of data involved in a neural network is huge. Then it is impossible to use an analytical approach. Therefore, an algorithm called gradient descent is used. For a simple initial step, a generic n-dimensional input array v is considered. For a small variation of each variable v_j the variation of the cost function is expressed in the following way:

$$\delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 + ... + \frac{\partial C}{\partial v_j}\Delta v_j + ... + \frac{\partial C}{\partial v_n}\Delta v_n \tag{2.9}$$

A more compact way can be written exploiting the concept of gradient of C:

$$\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T \Delta C \approx \nabla C \Delta v \tag{2.10}$$

Where Δv is the vector representation of the variations of v.

The gradient's notation directly shows how the variations of v are linked with the variation of $C(v)$. At this point, the purpose is to find a set of Δv such that ΔC is negative. Among all possibilities to justify this fact, a visual metaphor may be used. Considering a cost function as a deep valley and starting somewhere on its surface, the goal of the process is to reach the lowest point (see Figure 2.10).

Hence, it is necessary to choose the appropriate movements Δv to go down, this correspond with a negative ΔC.

Another way to explain it, requires the help of another parameter called *learning rate* defined as:

$$\Delta v = v^{'} - v = -\eta \nabla C \tag{2.11}$$

The representative equation of gradient descent can be obtained by combining the equations 2.10 and 2.11:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = \eta ||\nabla C||^2 \tag{2.12}$$

Therefore, an update rule for the parameters v is provided by the algorithm:

$$v \rightarrow v' = v - \eta \nabla C \tag{2.13}$$

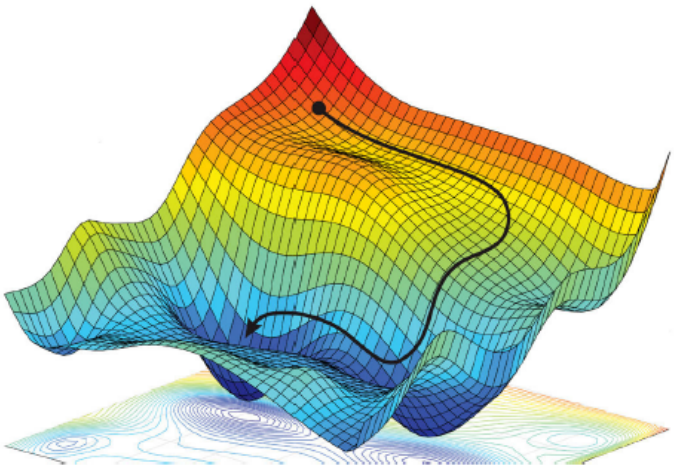


Figure 2.10: Gradient Descent visualization on a 3D surface

To summarise, by choosing a suitable set of changes in the parameters, the cost function can be minimised using the gradient descent technique.

The learning rate has a crucial importance, or better only if it is chosen correctly. In general, it has to be small enough to guarantee a good approximation of ΔC, especially when the final goal is almost reached and only fine adjustments are needed. On the contrary, when the global minimum is still too far and so $C(v)$ is not close to 0, it should not speed down the process too much. This is why, sometimes it is modified during the process according to an update rule in real time.

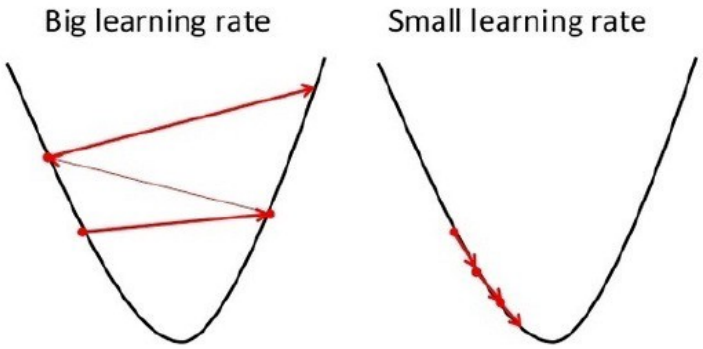


Figure 2.11: A comparison between a big learning rate and a small one using gradient descent algorithm

Finally, the update rule provided by the gradient descent algorithm is analysed devoting to the weights and biases of a neural network. The formula 2.14 , describe how an ANNs are actually trained. For a j^th weight w_j and bias b_j it looks like:

$$\begin{aligned} w_j \rightarrow w'_j &= w_j - \eta \frac{\partial C}{\partial w_j} \\ b_j \rightarrow b'_j &= b_j - \eta \frac{\partial C}{\partial b_j} \end{aligned} \tag{2.14}$$

2.7 Insight on training neural networks

Overfitting is well-known issue affecting neural networks. It is a concept in data science, which occurs when a statistical model fits exactly against its training data. In this case, the algorithm cannot perform accurately against unseen data when this happens, defeating its purpose. When machine learning algorithms are constructed, they leverage a sample dataset to train the model. However, it must be paying attention about how long the model is trained on sample data or considering how complex is the model under investigation. It is enough that one of the two arise that the system starts to learn the "noise," or irrelevant information, within the dataset. When the model memorizes the noise and fits too closely to the training set, it becomes "overfitted", and cannot generalise new data well. Therefore, a model that cannot generalize well to new data, it will not be able to perform the classification or prediction tasks that it was intended for. Low error rates and high variance are good indicators of overfitting. In fact, to check overfitting these parameters have low error rate on the training data and high error rate on the test data. A first way to prevent this behaviour is typically to set aside as the "test set" part of the training dataset [20].

Overfitting must be strongly avoided in Machine Learning, since a neural network able to provide good results only with data contained in the training set is useless. As previously stated, overtraining a model or having a model complexity to heavy lead in overfitting. Firstly, a logical prevention response could be either to pause training process earlier ("early stopping"), or to reduce complexity in the model by deleting inputs that are not relevant for the application accounted for. However, pausing too early or exclude too many important features the model can be underfitted encountering the opposite problem (Figure 2.12). In both cases, the model is not able to establish the dominant trend within the training dataset. Underfitting, as well as overfitting generalizes poorly to unseen data; however, a model that is underfitted presents high bias and less variance within their predictions. This illustrates the bias-variance tradeoff in Figure 2.13, which occurs when as an underfitted model shifted to an overfitted state. As the model learns, its bias reduces, but it can increase in variance as becomes overfitted. When

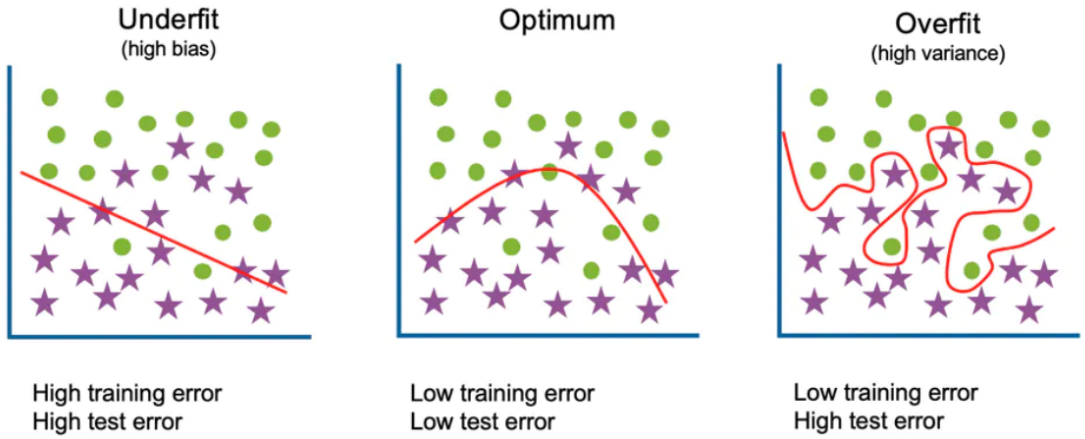


Figure 2.12: Overfitting vs Underfitting [20]

fitting a model, the goal is to find the "sweet spot" in between underfitting and overfitting, so that it can establish a dominant trend and apply it broadly to new datasets.

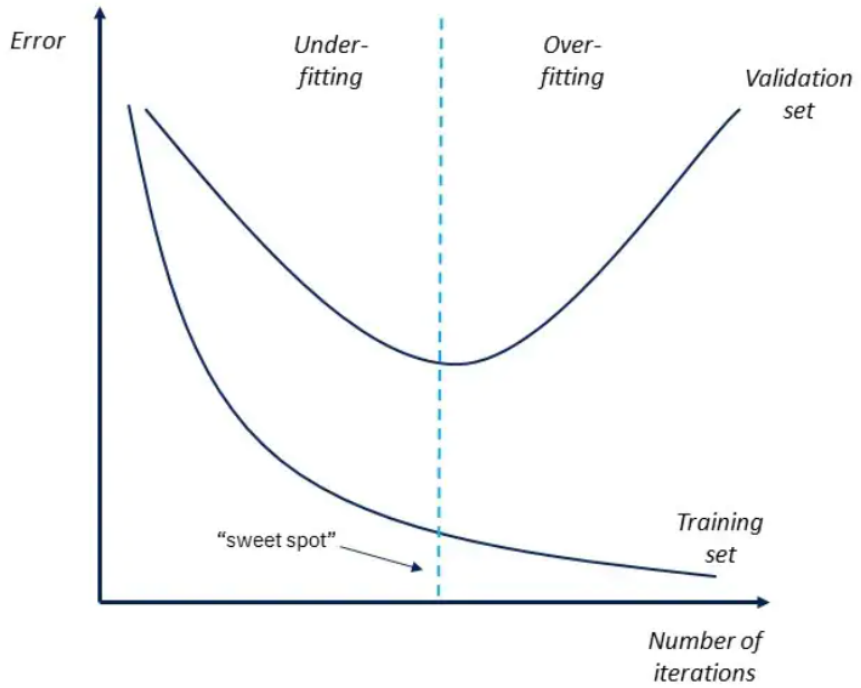


Figure 2.13: Error in function of number of iterations. The bias-variance tradeoff [20]

The ability to generalize the performance on different data can be improved with several methods. Below are a number of techniques that can be use to prevent overfitting:

Splitting data

A first solution might be splitting the available data in three subsets. In this case, the new training dataset will contain only a subgroup of the original dataset used for the proper learning process. Then, a validation dataset will be accounted for monitoring the performance of the network at the end of each epoch. It is fundamental since overfitting can be detected by looking at the accuracy gap between these two datasets. Finally, the network is tested with the Test dataset.

Early stopping

This method is based on pausing the training before it reaches the end. As already mentioned, this process risks halting the training process too soon, leading the problem of underfitting. Finding what is called the "sweet spot" is the final goal of this possible solution.

Train with more data

Expanding the training set can increase accuracy of the model giving more possibilities to parse out the dominant relationship between the input and the output variables. However, it is fundamental to pay attention to do not add more complexity to the model causing it to overfit.

Data augmentation

The availability of a huge dataset is only guaranteed in Machine Learning. Hence, it is not easy to train a deep neural network with a considerable number of parameters. This solution is one of the most used in particular cases such as image classification. In this way, it is possible to expand the training dataset with artificial samples with a set of different transformation, for example rotation, cropping, flipping or filtering create new artificial images. It is useful when the number of samples for each class in the training set is unbalanced.

Feature selection

Sometimes, it is possible to have some parameters or features used to predict a model that are redundant to each others. Feature selection is the process of

identifying the most relevant ones within the training data and eliminating the redundant ones.

Regularization

If overfitting happens when a model is too complex, it is possible to fix it reducing the number of features. However, it is used when the features useless in the model are not known and so the Feature selection cannot be applied. Regularization applies a penalty to the input parameters with the larger coefficient limiting the amount of variance in the model. A considerable number of regularization methods are used nowadays such as L1 regularization, dropout, and Lasso regularization. they are devoting to reduce the noise within the data.

2.8 Convolutional Neural Networks (CNN)

In the previous sections, artificial neural networks have been explained. However, it has to depict that the architecture taken into account only involves fully-connected layers where each neuron in the hidden layer is connected to all the neurons of the following layer and the previous one. However, this architecture is not very efficient when dealing with input as images. The reason can be addressed by two reasons: firstly, the network has a huge number of parameters to train making the computational complexity too expensive. Secondly, with this configuration the spatial structure of the image is not considered. To fix those problems the Convolutional Neural Networks (CNNs) are introduced. In general, the architecture of CNN are optimized for visual based tasks mainly devoted to image classifications. The purpose of this paragraph is to give a simple explanation of how the convolutional operations actually works in CNN and why it is efficient. The three pillars of CNN can be identified in the following concepts:

- Local receptive field
- Shared weights
- Pooling

The solution is to associate each neuron to a small region of the image, aiming to avoid a full connection between input pixels and neurons of hidden layers. This is called the *local receptive field* of the neuron. It learns a weight for each connection and a unique general bias. Hence, each receptive field region in the input image is connected to a neuron of the first neighbouring layer. By sliding the local receptive field by one pixel (or by a general quantity called stride), a connection with the second neuron of the hidden layer is created [21]. This operation is repeated until

the completeness of the input images. Moreover, all these neurons share the same weights and biases. The resulting number of neurons in the hidden layer will be:

$$n_h = \frac{W - F - 2P}{S} + 1 \tag{2.15}$$

W is the image width, F is the receptive field size, S is the stride, and P is the zero-padding. Better control on the output size of the layer can be obtained by setting to 0 the pixels along the image border. Assign with σ as a generic activation function and with $a_{x,y}$ the input activation function at position x,y, the output of a j,k-th hidden neuron will be:

$$out_{j,k} = \sigma\left(b + \sum_l \sum_m w_{l,m} a_{j+l,k+m}\right) \tag{2.16}$$

It implies that a single hidden layer can learn a single feature in the input image,

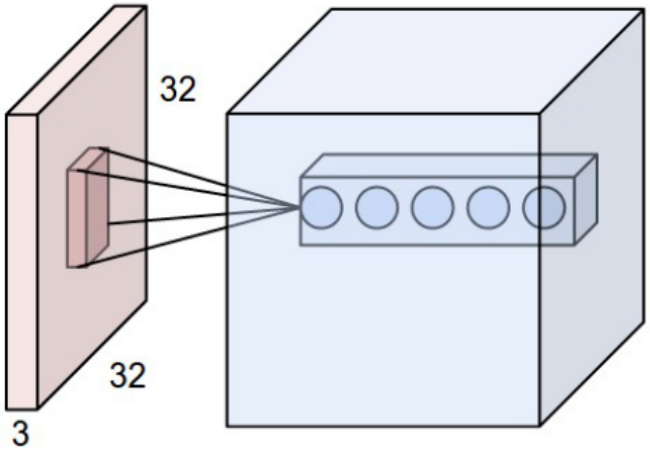


Figure 2.14: An example of input 3 channel image 32x32x3 mapped to a first layer with 5 feature maps [21]

at different locations. So, in CNNs the map from input to hidden layer is called *feature map*. The shared weights and bias together identify a kernel or filter. As it is immediate to see, since the weights are shared, the number of total parameters used in the network is dramatically reduced concerning the fully connected architecture allowing a faster training process. The feature map number is a design parameter for the convolutional layer that depends on both the task and the input image.

A convolutional layer is often associated with a pooling layer. The pooling is an operation performed directly on small input regions to simplify further the information contained.

2.9 Computer Vision

Computer Vision is an interdisciplinary scientific field that deals with how computers can gain high-level understanding from digital images or videos. Computer vision tasks include methods for acquiring, processing, analysing and understanding digital images, and extracting high-dimensional data from the real world to produce numerical or symbolic information, e.g., in the forms of decisions [22][23]. The scientific discipline of computer vision is concerned with the theory behind artificial systems that extract information from images. The images data can take many forms, such as video sequences, views from multiple cameras or multi-dimensional data from a 3D scanner [24]. Concerning the different ways to take images, this thesis project has been done by using a stereo camera to implement one of the sub-domains of computer vision, such as object detection.

2.9.1 Object Detection

Object detection is a computer technology related to computer vision and image processing that deals with detecting instances of semantic objects of a certain class in digital images and videos [25]. In general, neural techniques are able to do end-to-end object detection without specifically defining features and are typically based on Convolutional Neural Networks (CNNs). Object detection aims to locating and classifying existing objects in one image, and labelling them with rectangular bounding boxes together with the confidences of existence.

As stated in [26], the problem definition of object detection is to determine where objects are located in a given image (object localization) and which category each object belongs to (object classification). So, the pipeline of a traditional object detection models can be mainly divided into three main stages:

- *Informative region selection*: since objects can be at any position and can have different size of ratios, an easy choice is to scan a whole image with a multi-scale sliding window;
- *Feature extraction*: to recognise different objects, there is the need to extract visual features which can provide a semantic and robust representation;
- *Classification*: it is needed to distinguish a target object from all the other categories.

However, only small gains have been obtained during the years using this method. The reason can be addressed in general by two main reasons: first, the generation of bounding boxes with a sliding window is redundant, inefficient and inaccurate; secondly, the use of trained shallow models.

In general, the frameworks of a generic object detection can be categorised into two types. The first one generates a region proposal at first and then classifies each proposal in different object detections; it follows a traditional object detection technique. The second one regards object detection as a regression or classification problem, adopting a unified framework to achieve final results such as categorises and locations in a direct way. Among the regression/classification based method, there are the two algorithms chosen for this project. In the next sections they will be presented deeply, making a comparison from a theoretical point of view.

2.10 SSD: Single Shot MultiBox Detector

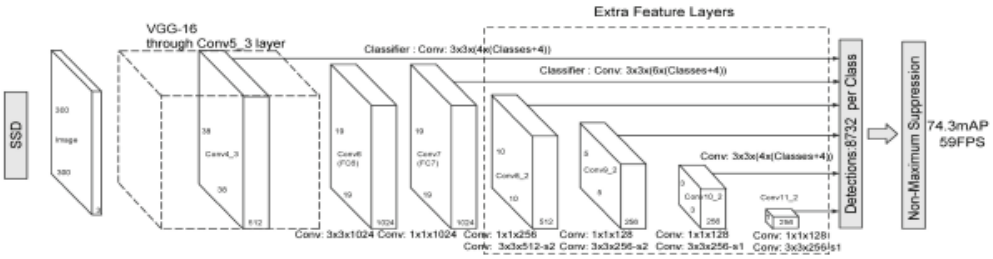


Figure 2.15: Single Shot Multibox Detector model: SSD adds several feature layers to the end of VGG16 backbone network to predict the offsets to default anchor boxes and their associated confidences. Final detection results are obtained by conducting NMS on multi-scale refined bounding boxes [27]

The SSD approach is based on a feed-forward convolutional network that produces a fixed-size collection of bounding boxes and scores for the presence of object class instances in those boxes, followed by a non-maximum suppression step to produce the final detections. The early network layers are based on a standard architecture used for high quality image classification (truncated before any classification layers), named base network (VGG-16) [27]. Then, additional structures are added to produce detections with the following key features:

- **Multiscale Feature Maps for Detection**: this is at the end of the base network. It decreases in size progressively and allow predictions of detections at multiple scales;
- **Convolutional Predictors for Detection**: they are identified on top of the 2.15. For a feature layer of size m x n with p channels, the basic element for predicting parameters of a potential detection is a 3 x 3 x p small kernel that produces either a score for a category, or a shape offset relative to the

default box coordinates. At each of the m x n locations where the kernel is applied, it produces an output value;

- **Default Boxes and aspect Ratios**: a set of default bounding boxes are associated with each feature map cell for multiple feature maps at the top of the network. The default boxes tile the feature map in a convolutional manner, so that the position of each box relative to its corresponding cell is fixed. At each feature map cell, the offset relative to its corresponding cell is predicted, as well as the per-class scores that indicate the presence of a class instance in each of those boxes. Specifically, for each box out of k at a given location, c class scores is computed and the 4 offsets relative to the original default box shape. This results in a total of $(c+4)k$ filters that are applied around each location in the feature map, yielding $(c+4)kmn$ outputs for a m x n feature map. For an illustration of default boxes, please refer to 2.16.

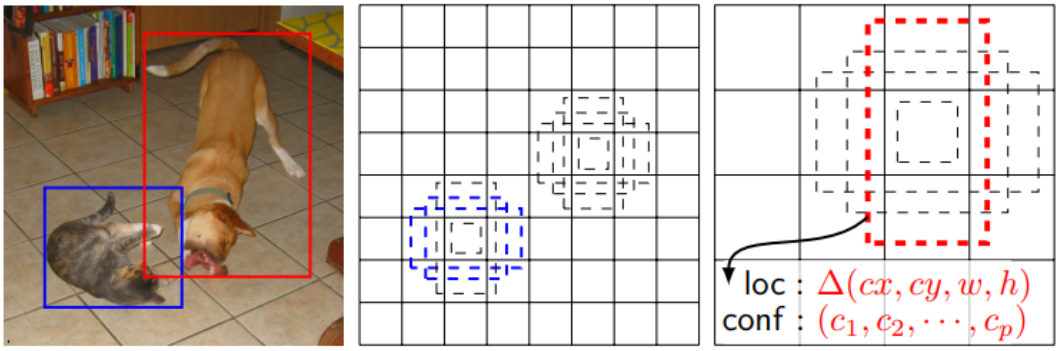


Figure 2.16: SSD framework, an example

2.10.1 MobileNet

The MobileNet network architecture is a special class of convolutional neural models that are built using a depth-wise separable convolutions and are not heavy in terms of their parameter count and computational complexity. Additionally, the authors involved in the development of this network architecture introduced 2 additional global hyper-parameters. These are the width and the resolution multiplier which can control the number of input/output channels of the convolution layers and the input data resolution (i.e Height, Width) respectively. These parameters can be used to directly influence the latency vs accuracy of the network depending on the end requirements of the user [28]. The integration of MobileNet into the

SSD framework is not an easy task as the Google research team has explained [29] evaluating the speed, memory and accuracy trad-off concerned with adapting diverse base feature extractors such as VGG16 and MobileNet within various detection architecture on different hardware and software platforms. It was shown that looking toward the SSD framework, the VGG backbone was a bottleneck during training and inference leadind to search another candidate. It was found in the MobileNetv1.

MobileNet is made up of Depth-wise Separable Convolutional layers that are computationally faster than standard convolutional layers. The reason is simply due to fewer *mult-adds* (multiplication and addition operations) due to the separation of channels in the depth-wise layer and their subsequent linear combination using the 1x1 convolution as shown in Figure 2.17 below.

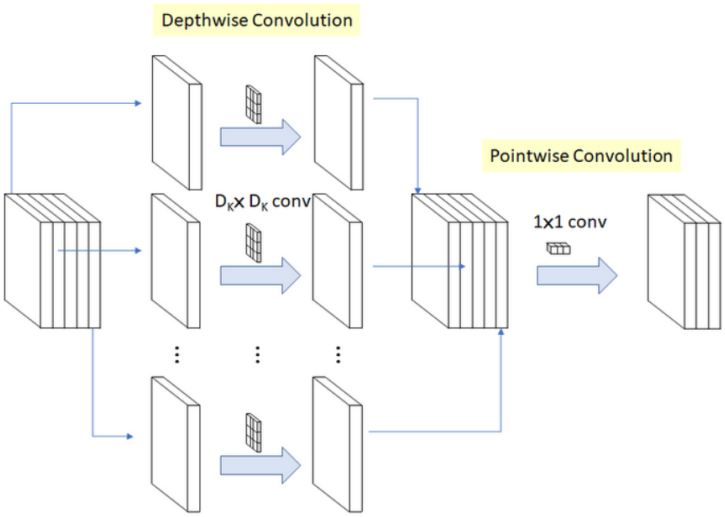


Figure 2.17: Depth-wise convolution

Empirically, it is shown that the reduction in computational effort does not affect the performance of the network to a large extent which is why the MobileNetv1 is a good choice as a backbone in the SSD framework. The full MobileNetv1 CNN architecture can be summarized as in Figure 2.18:

However, since it is used just as backbone the last three layers (AVG Pool, Fully connected, Softmax) are dropped out and replaced with the one proper by the SSD framework.

Type	Stride	Kernel shape	Input size
Conv	2	3 × 3 × 3 × 32	224 × 224 × 3
Conv DW	1	3 × 3 × 32	112 × 112 × 32
Conv	1	1 × 1 × 32 × 64	112 × 112 × 32
Conv DW	2	3 × 3 × 64	112 × 112 × 64
Conv	1	1 × 1 × 64 × 128	56 × 56 × 64
Conv DW	1	3 × 3 × 128	56 × 56 × 128
Conv	1	1 × 1 × 128 × 128	56 × 56 × 128
Conv DW	2	3 × 3 × 128	56 × 56 × 128
Conv	1	1 × 1 × 128 × 256	56 × 56 × 128
Conv DW	1	3 × 3 × 256	28 × 28 × 256
Conv	1	1 × 1 × 256 × 256	28 × 28 × 256
Conv DW	2	3 × 3 × 256	28 × 28 × 256
Conv	1	1 × 1 × 256 × 512	14 × 14 × 256
Conv DW	1	3 × 3 × 512	14 × 14 × 512
Conv	1	1 × 1 × 512 × 512	14 × 14 × 512
Conv DW	1	3 × 3 × 512	14 × 14 × 512
Conv	1	1 × 1 × 512 × 512	14 × 14 × 512
Conv DW	1	3 × 3 × 512	14 × 14 × 512
Conv	1	1 × 1 × 512 × 512	14 × 14 × 512
Conv DW	1	3 × 3 × 512	14 × 14 × 512
Conv	1	1 × 1 × 512 × 512	14 × 14 × 512
Conv DW	1	3 × 3 × 512	14 × 14 × 512
Conv	1	1 × 1 × 512 × 512	14 × 14 × 512
Conv DW	2	3 × 3 × 512	14 × 14 × 512
Conv	1	1 × 1 × 512 × 1024	7 × 7 × 512
Conv DW	2	3 × 3 × 1024	7 × 7 × 1024
Conv	1	1 × 1 × 1024 × 1024	7 × 7 × 1024
AVG Pool	1	Pool 7 × 7	7 × 7 × 1024
Fully Connected	1	1024 × 1000	1 × 1 × 1024
Softmax	1	Classifier	1 × 1 × 1000

Figure 2.18: MobileNetv1 CNN architecture [28]

Generally speaking, the role of the backbone network in the SSD framework is to convert the pixels from the input image into features that describe the contents of the image, and pass these along to the other layers of the SSD. Hence, it is used here as a feature extractor for a second neural network. Morevoer, it was already introduced that MobileNet has two tunable hyperparameters that allow the tradeoff between accuracy and computation. The width multiplier which weights the input and output channels and resolution multiplier which weights the input and output resolution. They reduce dramatically the computational cost [29].

2.11 YOLO: You Only Look Once

In this section it is given to the reader a brief background about all the object detection models of the YOLO family. Ever since the first YOLOv1 was introduced in 2015, multiple versions of YOLOv2, YOLOv3 and YOLOv5 have been released by different research groups [30].

Firstly, since every ML-based are evaluated in the base of their accuracy and robustness, let's introduce some parameters:

- **Recall** is the ratio of true positives to total positive prediction(correct or incorrect);
- **Precision** is the ratio of true positives to the ground truth positives(total correct predictions);
- The mean of all average precision is called **mean average precisions(mAP)**.

Thus, the YOLO creator tried to implement an object detection aiming to maximizes mAP. In general, the architecture of all the YOLO models have a similar theme that are:

1. **Backbone**: A convolutional neural network that accumulates and produces visual features with different shapes and sizes. Classification models like ResNet, VGG, and EfficientNet are used as feature extractors.
2. **Neck**: A set of layers that integrate and blend characteristics before passing them on to the prediction layer. Example: Feature pyramid network(FPN).
3. **Head**: Takes in features from the neck along with the bounding box predictions. Performs classification along with regression on the features and bounding box coordinates to complete the detection process. Outputs 4 values, generally x, y coordinates along with width and height.

There are two types of object detection models, one stage or two stage models. A one stage model is capable of detecting objects without the need for a preliminary step. On the contrary, a two stage detector uses a preliminary stage where regions of importance are detected and then classified to see if an object has been detected in these areas. The advantage of a one stage detector is the speed it is able to make prediction quickly allowing real time use [31].

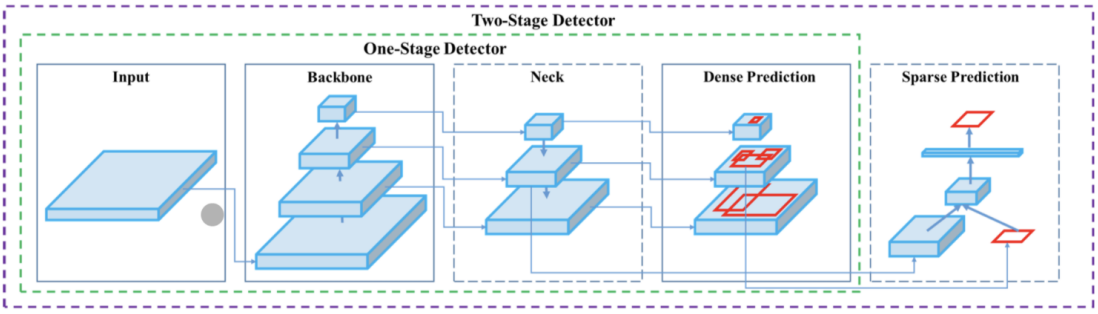


Figure 2.19: Object detection architecture showing both the one stage and two stage models

YOLOv1

The first YOLO model was introduced by J. Redmon in 2015 [32]. His pa-

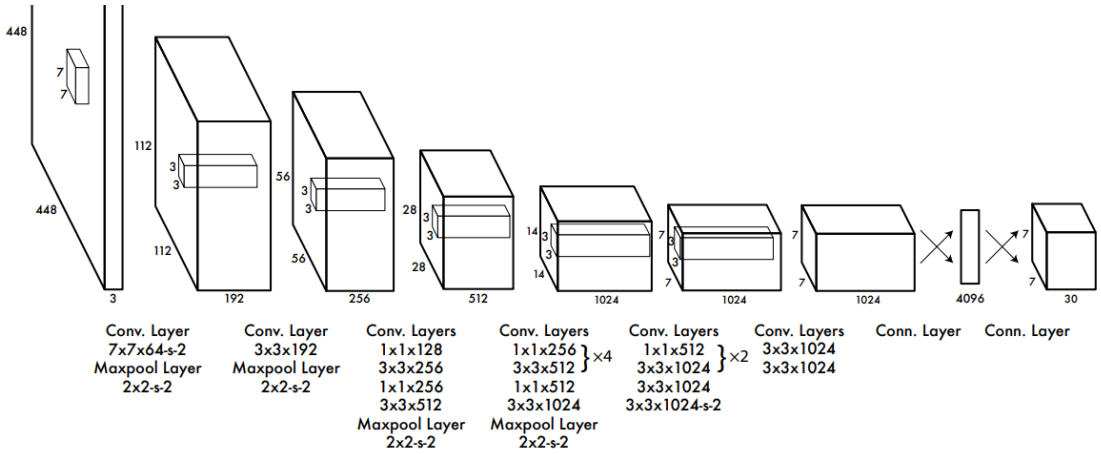
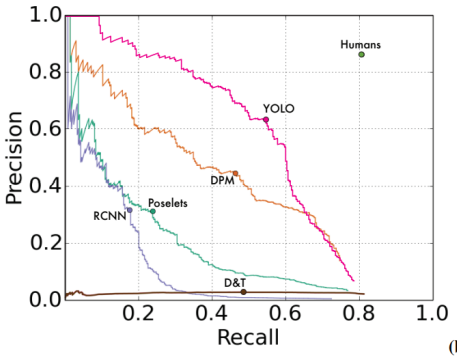


Figure 2.20: The Architecture

per was revolutionary in the object detection field since it started to replace the RCNN model which is accurate but slow in time due to its configuration done by multi-steps. In fact, firstly it finds the proposed region for the bounding box, then do classification over these regions and finally do post-processing to refine the output. So, the main YOLO objective is to replace the multistage perform object detection in just a single stage, increasing the inference time. In relation to the performance obtained, it is possible to see that YOLOv1 sported a 63.4 mAP with 45 frames per second speed, which means 22ms per image. In comparison, the speed inference rates for the RCNN ranged between 143ms to 20 seconds.



(a) Picasso Dataset precision-recall curves.

	VOC 2007	Picasso		People-Art
	AP	AP	Best F_1	AP
YOLO	**59.2**	**53.3**	**0.590**	**45**
R-CNN	54.2	10.4	0.226	26
DPM	43.2	37.8	0.458	32
Poselets [2]	36.5	17.8	0.271	
D&T [4]	-	1.9	0.051	

(b) Quantitative results on the VOC 2007, Picasso, and People-Art Datasets. The Picasso Dataset evaluates on both AP and best F_1 score.

Figure 2.21: Generalization results

The basic working principle of the YOLO model relies upon its unified detection tool, which groups into a single feed neural network different components of object detection. When an incoming image arrives in the model, it is divided into numerous grids and it is calculated the probability of an object resides inside that grid. This procedure is repeated for all the grids that the image is divided into. Then, the algorithm performs the task of grouping nearby high-value probability grids as a single object. Low-value predictions are deleted using a technique called Non-Max Suppression (NMS). Finally, the model is trained in a similar fashion where the centre of each object detected is compared with the ground truth. This is done to check the correctness of the model and to adjust the weights accordingly.

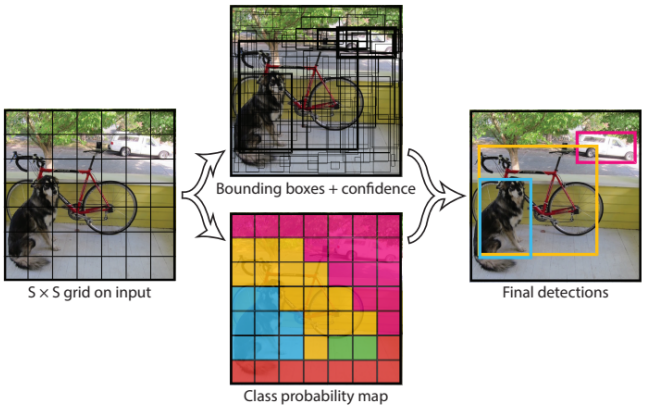


Figure 2.22: The Model. System models detection as a regression problem. It divides the image into an S × S grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities

YOLOv2

J. Redmon and A. Farhadi released this upgrade in 2016 through the paper named "YOLO9000: Better, faster, stronger" [33]. Together with various improvements made, the 9000 presented in the titled mean that it was able to detect over 9000 classes of objects. The performance had registered a huge improvement on the VOC 2012 dataset where reached 78.6 mAP. The major improvement from a technical point of view is the concept of the *anchor boxes*. Anchor boxes are the predefined area for an image that illustrates the idealised position of the objects to be detected. Moreover, an important parameter calculated as the ratio of overlap over union (IoU) areas acts as a threshold to decide if the probability of the detected objects is sufficient to make a prediction or not.

Anchor boxes are computed randomly in multiple object detections algorithms

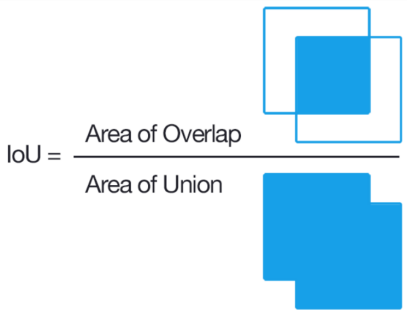


Figure 2.23: Graphical illustration of intersection over union (IoU) metric

except for the YOLO algorithm since it examines the training data and performs clustering on it (dimension clusters). This ensures that the anchor boxes used represent the data on which the model will be trained enhancing the accuracy.

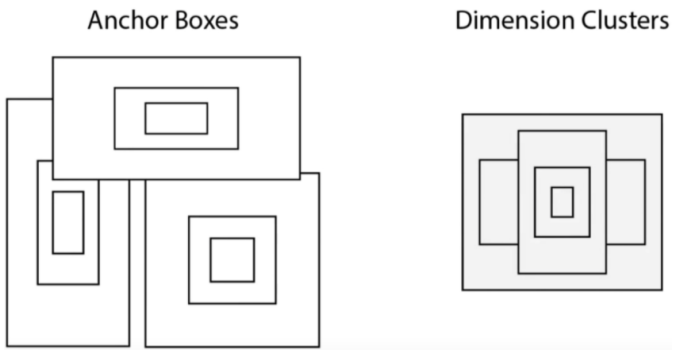


Figure 2.24: Anchor boxes converted to dimension clusters

There are additional improvements concerning the previous version. Firstly, in order to adapt to different aspect ratios, the YOLOv2 model is randomly resized throughout the training process. Further, to make the model robust was trained on a combination of COCO dataset (80 classes with bounding boxes) and the ImageNet dataset (22k classes without bounding boxes). Finally, using as a backbone a classification architecture called darknet19, as depicted in Figure 2.25, the inference speed has reached up to 200 FPS (frame per seconds) and mAP of 75.3.

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 2.25: Darknet19 Architecture

YOLOv3

It was enounced in 2018 through the paper "YOLOv3: An incremental Improvement" written by J. Redmon and A. Farhadi [34]. The model presented was a little

bigger than the earlier ones but much more accurate and yet fast enough.

The performance reached by the YOLOv3 can be seen in Figure 2.26. It shown a mAP of 28.2 with an inference time of 22 milliseconds using the COCO dataset. Those results highlight that YOLOv3 is three times fast than the SSD object detection technique obtaining the same accuracy.

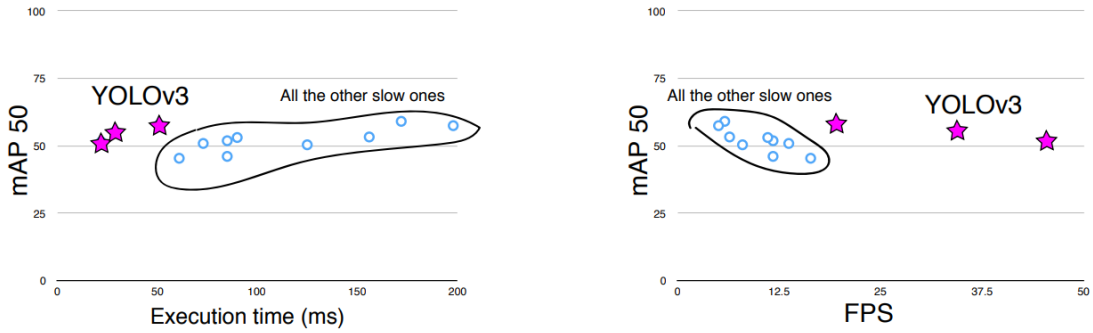


Figure 2.26: Yolov3 vs other slow algorithms

From a technical perspective, it consists of 75 convolutional layers without using fully connected or pooling layers, which consistently reduced the model size and weight. Furthermore, it uses as a feature extractor a feature pyramid network that is able to extract different types, forms, and sizes of features for a single image. It concatenates all the features is such a way that the model can learn local and general features. Finally, the YOLOv3 paper pointed out by using a logistic classifier and activations, the class prediction goes above and beyond RetinaNet-50 and RetinaNet-101 in terms of accuracy. Here, the backbone used for the YOLOv3 is the Darknet53 architecture.

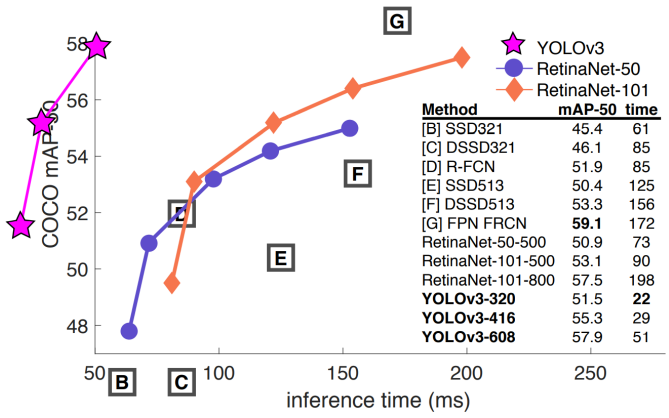


Figure 2.27: mAP versus Inference time

YOLOv4

On the contrary to all the other already released versions, YOLOv4 was not released by J. Redmon but by Alexey Bochkovskiy et al. in 2020 [35]. In Figure 2.28, the comparison among other detection models is presented, showing the good performance of YOLOv4. It has reached a speed of 62 Frame Per Seconds with a mAP of 43.5 percent on the COCO dataset.

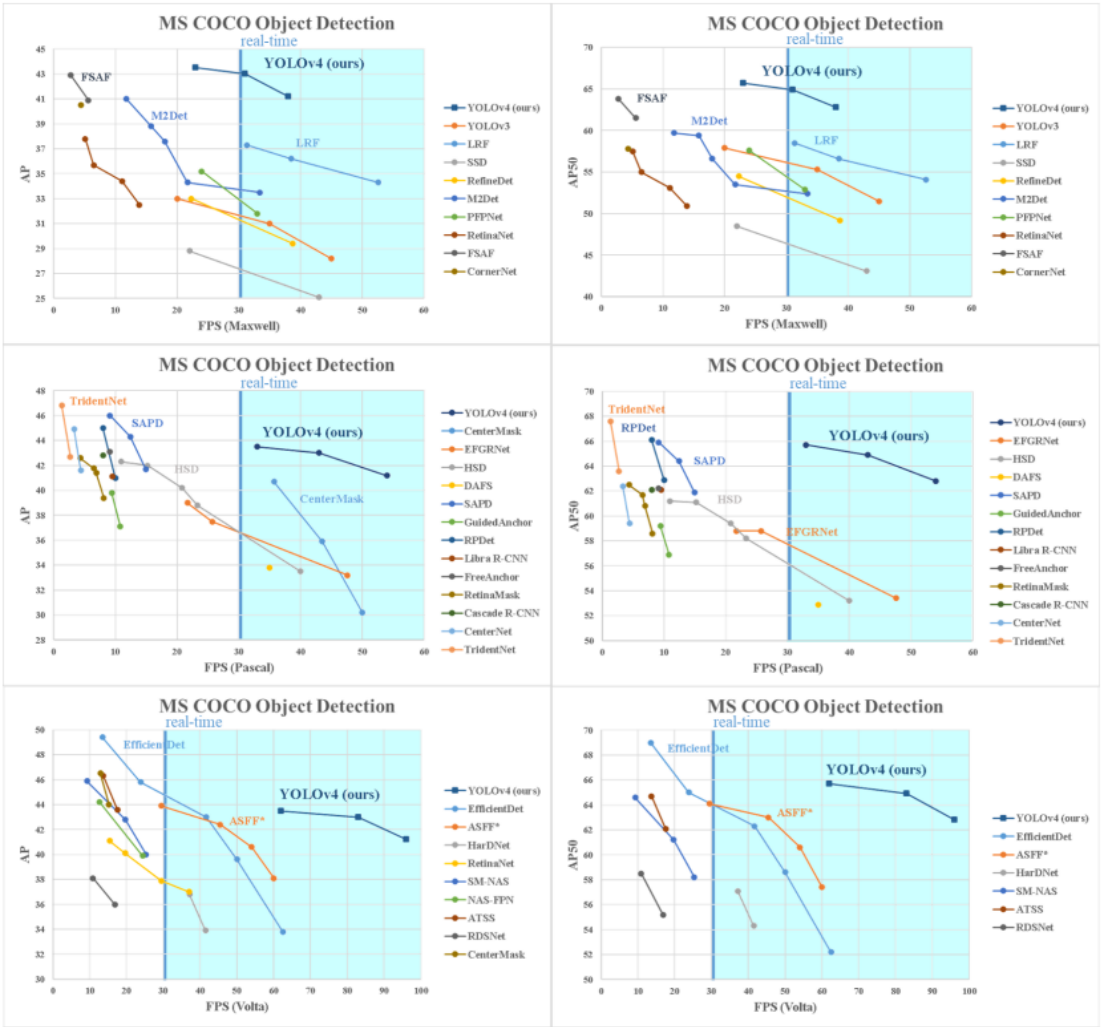


Figure 2.28: Comparison of the speed and accuracy of different object detectors

The technical improvements that YOLOv4 has been carried out are related to introducing two concepts called *bag of freebies* and *bag of specials*. The former presents some techniques that bring about an enhancement in model performance without increasing the inference cost. In contrast, the latter are techniques that

increase accuracy while increasing the computational cost. The main bag of freebies are:

1. **Data augmentation techniques**: Cutmix (Cut and mix multiple images containing objects that we want to detect), Mixup(Random mixing of images), Cutout, Mosaic data augmentation.
2. **Bounding box regression loss**: Experimentation of different types of bounding box regression types. Example: MSE, IoU, CIoU, DIoU.
3. **Regularization**: Different types of regularization techniques like Dropout, DropPath, Spatial dropout, DropBlock.
4. **Normalization**: Introduced the cross mini-batch normalization which has proven to increase accuracy. Along with techniques like Iteration-batch normalization and GPU normalization.

Instead, the bag of specials are:

1. **Spatial attention modules (SAM)**: Generates feature maps by utilizing the inter-spatial feature relationship. Help in increasing accuracy but increase the training times.
2. **Non-max suppression (NMS)**: In the case of objects that are grouped together we get multiple bounding boxes as predictions. Non-max suppression reduces false/excess boxes.
3. **Non-linear activation functions**: Different types of activation functions were tested with the YOLOv4 model. Example ReLU, SELU, Leaky, Swish, Mish.
4. **Skip-Connections** like weighted residual connections (WRC) or cross-stage partial connections(CSP).

Chapter 3

Hardware and Software Configurations

Before diving into the discussion of the proposed method, this chapter is dedicated to the presentation of the hardware and software configurations of the ADS system mounted on the prototype SC19. Perception, views as sensing and processing techniques, has a pivotal role in the complete perception pipeline of the ADS. The central topic of this chapter is the introduction of the sensors used to acquire the external measurements from the environment. Although the stereo camera is the main component used in this project, also the Inertial Measurement Units is explained. The reason lies in the final goal of this thesis since both of these two sensors are then implemented on a rapid platform for the validation stage. Finally, also the LiDAR is briefly explained. After presenting all the sensors used, the overall hardware architecture design is shown, highlighting all the networks between the sensors and the nodes. Finally, to understand what a node is, the Robot Operating System (ROS) environment is presented with all the three pillars that compose the Perception pipeline.

3.1 Sensors

Sensors are the fundamental components of a perception pipeline used in an autonomous vehicle. They cover an essential role for autonomous driving technologies development under broad aspects; however, they deserve a deep analysis in economic and technical points of view. A sensor is a mechatronic device able to detect specific properties of the physical environment under which it is exposed, processing this information and forward for the following processing phase. In general, it is composed of two main elements, a sensitive and a transducer element. The first one aims to interact with the external input, while the second one has to convert

the input measured in an output signal in a form that the acquisition system can read.

To perform the object detection task some sensors are needed. The ones that are able to see the environment surrounded by the vehicle are the Stereo camera and LiDAR, while the on-board computing platform makes all the computational effort. For this thesis purpose the LiDAR is not considered, so all the notions are coming from the camera. Obviously, using the two sensors through the sensor fusion guarantees better robustness of all the pipeline. The main sensors used by Squadra Corse PoliTo Driverless are presented in the following subsections, with a deeper analysis for the one used for the perception pipeline proposed.

3.1.1 Stereo camera

Stereo cameras are one of the most important sensors available for autonomous navigation because they allow computer-vision implementation. An image is a two-dimensional projection of three-dimensional space, it inherently causes a loss in depth information. However, if two images from different perspectives are available, and the cameras' intrinsic parameters and the relative position between each other is known a priori, it is possible to triangulate the position of a common point to get its three spatial coordinates. From a hardware point of view, a Stereo camera is a type of camera with two or more lenses with a separate image sensor or film frame for each lens. This allows the camera to simulate human binocular vision and, therefore, capture three-dimensional images [36]. This process is called Stereo Vision.

Pose estimation with Stereo Vision consists mainly in 4 steps represented in a schematic way in Figure 3.1:

- **3D Reconstruction and Calibration**: it consists of estimating the relative position between the two cameras and their relationship with the environment obtaining thanks to the calibration phase. Not only the extrinsic parameters but also the intrinsic ones can be achieved through the calibration stage, required to correct the camera lenses distortion and for the step of triangulation (Figure 3.1 a);

- **Feature detection and correlation**: it is necessary to identify points in an object that are visible in both pictures. This process is not straightforward since a point in an image could not necessarily be precisely identified in the other one (Figure 3.1 b);

- **Triangulation**: with the information of the intrinsic and extrinsic parameters of the cameras and the location of the feature points in both images, their position in space can be derived (Figure 3.1 c);

- **Pose estimation**: to define the pose of a rigid body it is necessary to determine six parameters since a single point is not enough. At least three points of the solid in space must be known to compute its position and orientation with respect to the world reference frame (Figure 3.1 d).

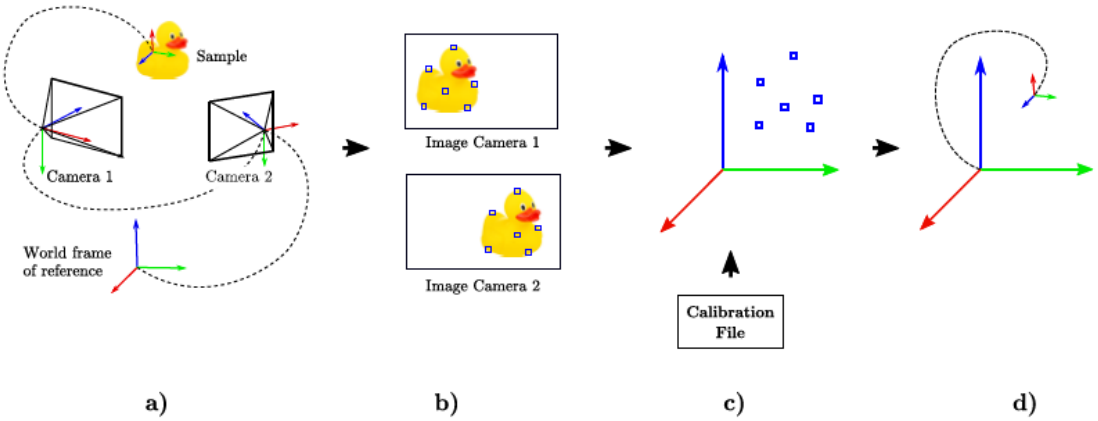


Figure 3.1: Stereo Vision overview

Stereo Calibration

The most used camera parameters that can be retrieved for each eye and resolution are:

- Focal length: fx, fy.
- Principal points: cx, cy.
- Lens distortion: k1, k2, k3, p1, p2.
- Horizontal, vertical and diagonal field of view.
- Stereo calibration: rotation and translation between left and right eye.

As already depicted previously, the reconstruction of the intrinsic and extrinsic parameters needs a stereo calibration stage. The goal is to find the relative position of the cameras to each other and to the world reference frame. In a more mathematical way, the objective is to find two homogeneous transformation matrices T_{c2}^{c1} and T_{c2}^{W}. the first transformation is from the camera 2 to camera 1, while the second one is from the camera 2 to the World reference system as it is depicted in Figure 3.2.

The extrinsic and intrinsic parameters can be computed at the same time. In general, the intrinsic camera parameters do not change much over time, on the

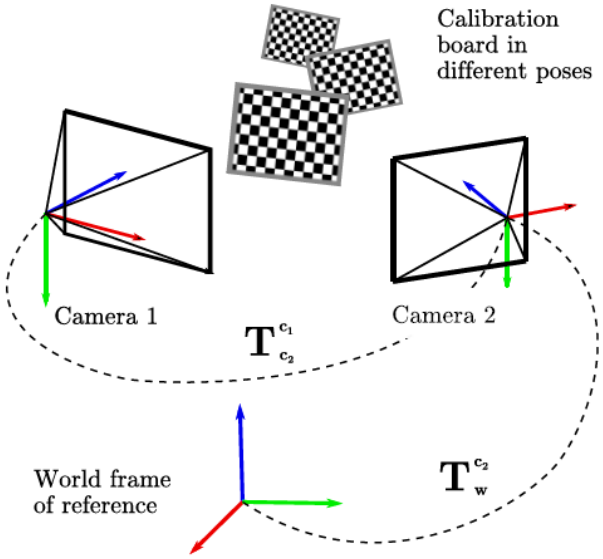


Figure 3.2: Reconstruction

other hand, the extrinsic parameters do if there is a slight variation on the relative pose of a camera with respect to the other. These changes in position could be caused simply by changes in temperature. Because the error should be reduced at its minimum expression, it is necessary to calibrate the extrinsic parameters frequently. A possible way to do calibration is to use a chessboard with the two cameras focusing on it, moving the board in different positions and taking images on it. The chessboard pattern is very useful for the computation of the intrinsic values since it covers almost the whole image. However, because of its dimensions, automating the calibration with such a board becomes impractical.

The camera used in the ADS proposed by Squadra Corse PoliTo Driverless is the ZED Stereo Camera from StereoLabs Company.

The dimensions of the camera are taken from the camera's datasheet (Figure 3.5).

Since the stereo camera is used for multiple functions, mainly for entertainment purposes, spatial analytic studies, and autonomous robotics, some configurations should be decided with respect to the application used. Thus, here the most important settings are explained, while in Section 4.1 will be analyzed the design choices that have been used.

First, a video mode has to be decided. The left and right video frames are synchronized and streamed as a single uncompressed video in syde-by-syde format. As shown in Figure 3.6, there are several video modes available. It is worth pointing out that decreasing the output resolution allows for a higher frame rate.

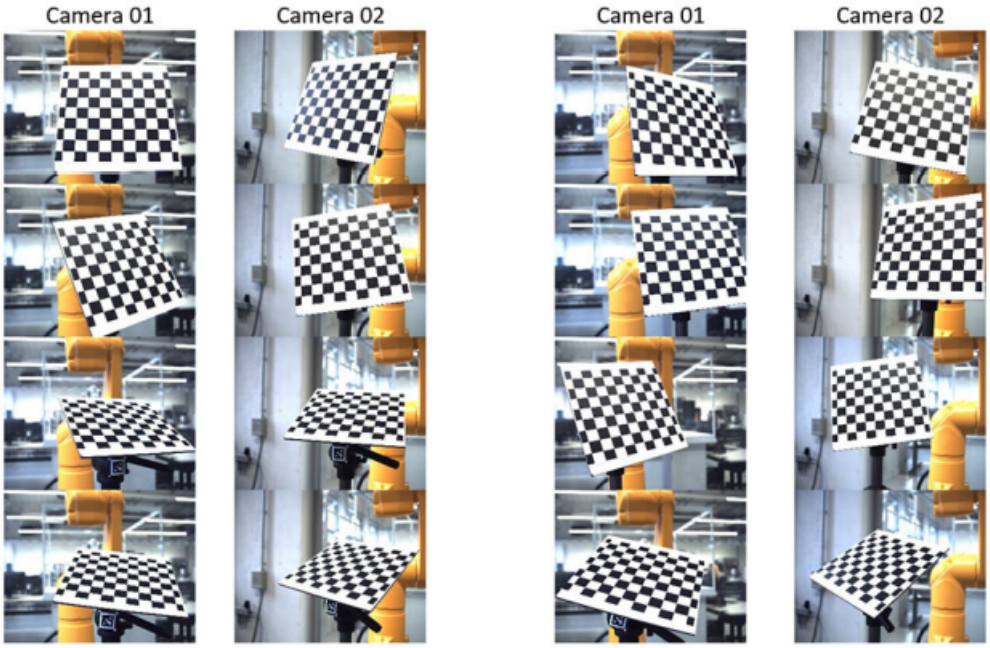


Figure 3.3: Example images for stereo calibration

Figure 3.4: Zed stereo camera

The ZED stereo camera reproduces the way human binocular vision works. Likewise, Stereolabs stereo cameras have two eyes separated by 6 to 12 cm which allow to capture high-resolution 3D video of the scene and estimate depth and motion by comparing the displacement of pixels between the left and right images [37]. The Depth Map captured by the ZED store a distance value (Z) for each pixel (X,Y) in the image. The distance is expressed in metric units and it is calculated from the back of the left eye of the camera to the scene object. The depth map is also displayed with a monochrome 8-bit representation where the values span from 0 to 255. The biggest values represent the closest possible depth value and the smallest one represents the most distant possible depth value. Furthermore, the depth has different depth mode to fit the application's need, they adjust the level

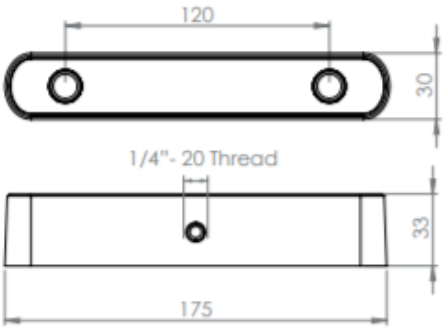


Figure 3.5: Physical dimensions of the camera taken from the datasheet

VIDEO MODE	OUTPUT RESOLUTION (SIDE BY SIDE)	FRAME RATE (FPS)	FIELD OF VIEW
2.2K	4416x1242	15	Wide
1080p	3840x1080	30, 15	Wide
720p	2560x720	60, 30, 15	Extra Wide
WVGA	1344x376	100, 60, 30, 15	Extra Wide

Figure 3.6: Video mode possibilities

of accuracy, range and computational performance of the depth sensing module:

- **ULTRA**: offers the highest depth range and better preserves Z-accuracy along the sensing range
- **QUALITY**: has a strong filtering stage giving smooth surfaces
- **PERFORMANCE**: designed to be smooth, can miss some details

The Stereolabs recommend the use of the ULTRA mode for both Desktop and embedded applications. However, if they require too many resources the Performance mode is suggested.

Finally, the depth range corresponds to the minimum and maximum distance at which the depth of an object can be estimated. Although the default distances could be enough, the company gives some tips to optimize the depth range. Lowering the minimum range to very small values can dramatically increase memory requirements

and reduce FPS (frame per seconds), so this value must be increased to have better performance. For applications requiring long-range depth perception, it is suggesting putting the depth minimum distance at least at 1m or more for improved performance.

DEPTH RANGE	ZED
Default	0.4m to 25m (1.3 to 82ft)
Min range	0.2m (0.6ft)
Max range	40m (131ft)

Figure 3.7: Depth range

3.1.2 Other Sensors

Inertial Measurement Units (IMU)

The Inertial measurement Units (IMUs) is by far one of the most common type of inertial sensors used thanks to their good performance, low weight and low cost. Depending on the specific application, nowadays a wide variety of IMUs are available. In general, all IMUs measure and report the body's specific forces, angular rates, and orientation, using a combination of accelerometers, gyroscopes, and eventually magnetometers. All these components are described to give to the reader a brief overview of the inertial sensors.

- *Accelerometers*: thanks to the accelerometers it is possible to measure the relative acceleration with respect to an inertial reference frame. Internal elements of an accelerometer can be modelled as a damped mass on a spring system: when the sensor is exposed to an acceleration, the mass displacement is measured by transducer with capacitive or piezoresistive effects;
- *Gyroscopes*: they are devices used to measure angular rates relative to an inertial reference frame. The ones building in the last years are the optical and the vibrating gyroscopes;
- *Magnetometers*: they are devices used to measure the local magnetic field by using the Hall effect. From a hardware point of view, they are made

by a thin sheet of semiconducting materials. They are used to derive the vehicle's heading by comparing the intensity of the measured field with the local intensity of Earth's magnetic field. Unfortunately, magnetometers have a very poor application on mobile robots and vehicles due to nearby metallic parts, electrical cabling, and power electronics systems.

Inertial Navigation System (INS)

Inertial Navigation Systems (INSs) represent the most common application of IMUs in the field of navigation. An Inertial Navigation System is a navigation device that uses a computing platform, multiple motions (accelerometers) and rotation sensors (gyroscopes). These components are used by dead-reckoning approach to continuously calculate the position, orientation, and velocity of a moving object without the need for an external reference. Often the inertial sensors are supplemented by a barometric altimeter and occasionally by magnetic sensors (magnetometers) and/or speed measuring devices (ground truth sensors or odometers) [38]. The major difference with the class of the IMUs, lays in the presence of an embedded computer that uses the measurements from the inertial sensors and the supplementary ones to assess the vehicle's state increasing the effectiveness in inertial measurements and enhanced accuracy concerning the stand-alone IMUs. INSs have broad possibilities to be implemented in complex systems such as mobile robots and land and air vehicles. The main advantage of having an on-board computer able to evaluate real-time position help to increase the accuracy of the sensor's readings and to avoid drift of the position measurements.

Global navigation satellite system receiver (GNSS)

Global Navigation Satellite System (GNSS) is a generic term that refers to any global satellite-based system that can pinpoint a user's geographical location anywhere in the world. Each GNSS consists of the satellites (known as a constellation), ground control network, and user equipment (receivers). There are currently two fully operational GNSS systems: America's Global Positioning System (GPS) and Russia's Global Orbiting Navigation Satellite System (GLONASS). The European Galileo and Chinese BeiDou GNSSs are partially operational, and are expected to become fully "global" in their coverage within a few years. GNSS satellites remain in medium Earth orbit (MEO) and transmit coded signals containing both precise orbital details and, thanks to atomic clocks, a very stable and accurate timestamp. The ground control network keeps track of the relevant satellite constellation, monitoring a range of data such as satellite health and signal integrity. It also ensures satellites remain in the correct orbital configuration. Furthermore, the ground

control network determines precise satellite orbits, updates the satellite clock corrections and provides other information essential to determining user position, velocity and time (PVT) [39]. It uses satellites to give geospatial positioning to electronic receivers to determine their location (longitude, latitude, and altitude/elevation) with high precision. The signals also guarantee the electronic receiver to calculate accurately the current local time, which allows time synchronisation.

After this small introduction among the major navigation sensors, the main hardware chosen for localisation is an Inertial Navigation System (INS). INSs are self-contained, non-radiating, dead-reckoning navigation systems that provide dynamic information through direct measurements. If integrated with absolute location-sensing mechanisms (such as GNSS receivers), it can provide accurate information about the vehicle's position at the centimetre level. The sensors individuated by Squadra Corse PoliTo Driverless to address the Localization pipeline are Global Navigation Satellite System (GNSS) receiver and an Inertial Navigation System (INS), which embeds an Inertial Measurement Unit (IMU). The retained sensor is the *SBG Systems Ellipse-N* expressively designed for autonomous driving systems (Figure 3.8a). The Ellipse-N embeds a 64-bit microprocessor that can process a sensor fusion routine in real-time from measurements of multiple sources: in the case of Ellipse-N, the sources are a 3-axis IMU, a GNSS receiver, and a barometer. The sensor is small-sized and features an integrated dual-band antenna *GNSS receiver* (Figure 3.8b). The external antenna, mounted on a flat surface of the car, provides orientation angles and accurate GNSS position to the processing algorithm. It has been best suiting for dynamic environments and harsh GNSS conditions but can also operate in lower dynamic applications with a magnetic heading. The sensor is fully compatible with the computational platform used on-board on the considered vehicles. It supports both serial communication at high baud rates with the Linux machine running ROS, and the CAN bus communication with any real-time automotive ECU.

(a) The SBG Ellipse-N **(b)** The GNSS antenna

Figure 3.8: Localization sensors mounted on SC19

LiDAR

The output of a camera is much easier to interpret; however, camera-based sensors have some drawbacks for autonomous driving tasks under varying light and visibility conditions and with scenes with a high dynamic range, such as entering or exiting a tunnel. Therefore, LiDAR sensors represent a recent technology in which sending millions of light pulses per second is able to register the amount of light reflected back by any non-absorbing object or surface. Differences in laser return times and wavelengths can then be used to create a 3D high-resolution point cloud representing the surroundings. The result set of points that is obtained is called pointcloud.

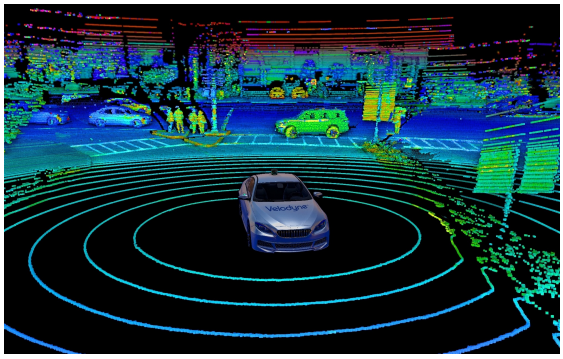


Figure 3.9: Example of LiDAR pointcloud in automotive field

The behaviour of a LiDAR, in this case, is unaffected. Although LiDAR sensors are mostly indicated for creating accurate 3D maps in a huge horizontal FOV (field of view), they also have some important drawbacks in the environment perception. They typically have a limited vertical resolution, and they are not suitable for detecting small objects placed at great distances since they compute a sparse map. Moreover, the LiDAR measurements could be strongly affected by light and weather conditions, so the usage of redundant camera sensors is suggested. To make robust perception pipeline against sensor failure, several methods have been proposed. For example, the deployment of parallel and independent sensing and estimation pipelines based on camera and LiDAR allows to get the most accurate information on the distance of the detected object and their visual features like colour. The LiDAR-based perception pipeline relies on data coming from a *Velodyne VLP-16* (Figure 3.10. The sensor can provide a full 360° view of the surrounding environment at 10 Hz to obtain an accurate real-time 3D data reconstruction recorded by 16 light channels. It ranges up to 100 m with 30° vertical FOV and an angular resolution up to 0.1° in the horizontal plane [40].

Figure 3.10: The Velodyne VLP-16

3.2 Hardware Architecture Design

The considered all-wheel drive electric vehicle is represented in Figure 3.11. The vehicle has an integral carbon fiber chassis built with honeycomb panels, double wishbone push-rod suspensions, an on-wheel planetary transmission system and a custom aerodynamic package. The vehicle can reach a maximum speed equal to 120 km/h with longitudinal acceleration peaks reaching up to 1.6 g [40]. In this section, all the hardware configuration implemented in the SC19 is presented starting from the sensors devoted to the perception task. The Velodyne VLP-16 LiDAR is positioned in the front wing of the vehicle at a height equal to 0.1 m from the ground. In this way the position is optimized for the limited vertical field of view of the LiDAR. The StereoLabs ZED stereo camera sensor is positioned to the vehicle's rollbar at a height of 1.05m. The NVIDIA Jetson AGX Xavier high-performance is placed inside the vehicle's monocoque. The NVIDIA is an embedded Linux high-performance computing platform with embedded GPUs with 32 Tera-Operations per Second (TOPS) of peak computational power and 750 Gbps of high-speed input/output capability in less than 50 W of needed power. Moreover, the NVIDIA's JetPack Software Development Kit 4.1.1 deployed for Jetson AGX Xavier includes CUDA 10.0, cuDNN 7.3, and TensorRT 5.0 libraries, providing a complete artificial intelligence software stack. The computing platform creates a ROS network, which allows to process the information streaming from the LiDAR and stereo camera sensors. ROS Melodic is used for the arm64 architecture of the computing platform that features Ubuntu 18.04 release [41]. A better understanding of the Robot Operating System (ROS) environment will be done in Section 3.3.1. Finally, proper cable connections has to be carried out to correctly interface and supply the computing platform together with the sensors. This role is done by a 12 V 10 Ah rechargeable Lithium battery as the devoted power source, along with properly connected DC-DC power converter for the high-performance computing

platform.

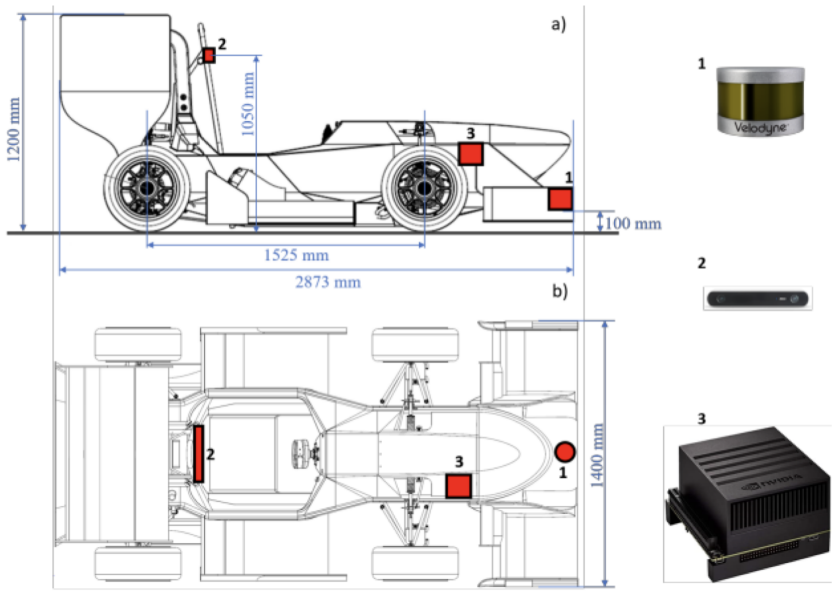


Figure 3.11: Vehicle layout with hardware positions

3.3 Software Architecture Design

After analysing the configuration of the sensors from a hardware point of view, the overall architecture of the autonomous system adopted by the first prototype SC19D by Squadra Corse PoliTo Driverless is described in Figure 3.12.

The system is generally composed of the on-board computing platform devoted to the main algorithms for autonomous driving, such as Perception, Localization, Mapping and Path Planning. It communicates with the main ECU – a dSpace MicroAutoBox II - used in the SC19D for the Motion Control task in a real-time fashion. The dSpace used is a real-time system for performing fast function prototyping that can also operate without user intervention. Specifically in the autonomous driving scenario, the control algorithm starts receiving as input the decision-making done by the autonomous control system running on the dSpace, performs some safe analysis, and delivers a torque request to the motors inverters controller. However, the computational power of an embedded computer is needed due to multiple causes such as environment perception allowing the cone detections and building the track map. This role is covered by the already present on-board platform NVIDIA Jetson AGX Xavier featuring with Linux Ubuntu 18.04 and Ros Melodic, which has been chosen for its computational power, embedded high

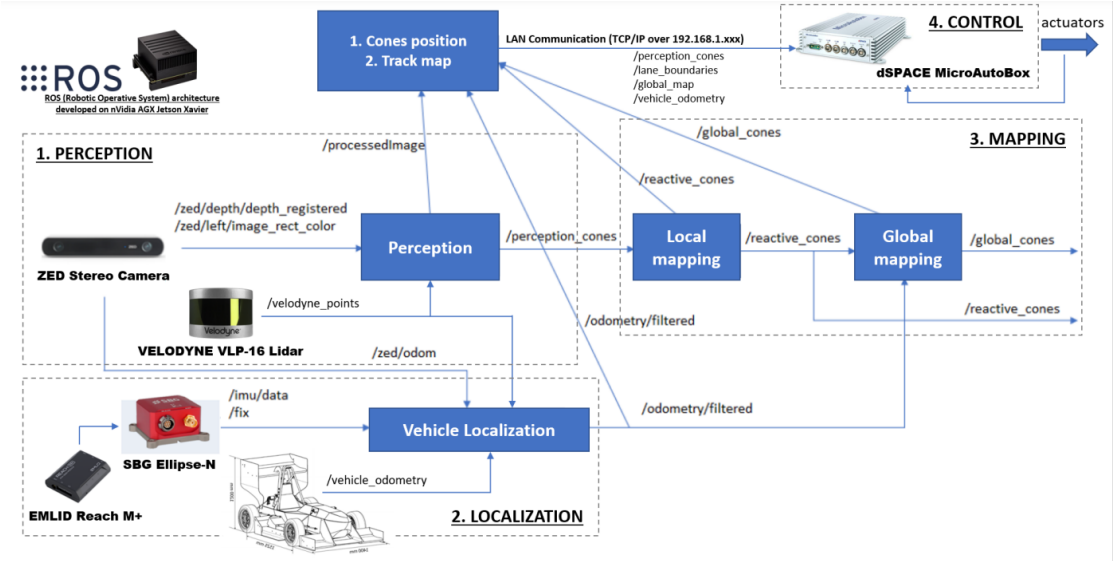


Figure 3.12: Overall architecture of Squadra Corse Driverless Autonomous Driving System

performance GPU unit (suited for image processing), and compact design. The NVIDIA onboard computer hosts the main pipelines of the autonomous stack:

1. **Perception**
2. **Localization**
3. **Mapping**

The correct behaviour of the whole pipeline and proper memory allocations are guaranteed by the Robot Operating System (ROS), running on the Linux Ubuntu 18.04 machine. In Section 3.3.1 more information about the ROS environment will be given together with an analysis of all the components belonging to the autonomous overall pipeline. Before explaining the three main autonomous tasks analysing what they do and how they communicate together, it is worth introducing the SLAM process to understand the localization and mapping goal better. The SLAM (Simultaneous Localization and Mapping) is a process through which a mobile robot, moving into a specific environment, is able to derive a map from its perception and simultaneously computes its own position within this map. Given these premises, it is easy to understand the complexity of this problem, especially compared to the two main tasks taken singularly:

- *Localization problem*: the complexity lies on the unknown map, thus the absence of a priori information about the environment.

- *Mapping problem*: the complexity lies in the lack of information on the pose.

In order to build a map from the environment the robot must be equipped with sensors that allow it to perceive and obtain measurements of the elements from the surrounding world [42]. However, SLAM is based on various hardware and software possibilities and different combinations of the two where one of the main constraints in mobile robotics is to use low-power, lightweight equipment and acceptable energy resources. Although there are some consistent differences among the SLAM algorithms, there is a quite common factor to all those algorithms, i.e. the development platform ROS. The following subsection will describe its basics.

3.3.1 Robot Operating System (ROS)

Although the name may suggest it, ROS is not correctly an operating system, but it consists of a collection of libraries, tools, and conventions aiming to simplify the task of creating robust and complex robot behaviour across a wide variety of robotic platforms. It provides the general services expected from a standard operating system, including hardware abstraction, low-level device control, implementation of commonly used functionalities, message-passing between processes, and package management. Different from a standard operating system (OS), it is extremely lightweight and oriented to rapid-prototyping of software components and software development. It is considered as a robotic middleware, so software that connects different software components and applications [43]. At the core of ROS are four different concepts, pointing out its philosophy:

- *Plumbing*: means that different programs can run simultaneously, and ROS enables the communication between different pieces of software. ROS acts as a "plumbing" in the sense that it provides the device drivers needed for communication between software and hardware;

- *Tools*: the second core part of ROS is the set of tools that it provides. Many basic tools are already implemented by ROS, being it visualisation, simulations, GUIs, or basic tools for data logging;

- *Capabilities*: capabilities can be thought of as high-level tools. They are software that can be installed on ROS and enable mapping, localization, planning, etc. These capabilities are provided by the Open Source community and enable the research teams to focus on single areas while being able to rely on the state-of-the-art solution to be customised to their specific needs;

- *Ecosystem*: being the de facto research standard, ROS boasts vast documentation, excellent compatibility, and tutorials provided for most of its software.

The inner-workings of ROS can be best understood by looking at its basic components:

- *Nodes*: processes that perform computation. A robot control system usually comprises many nodes. Nodes are used for different purposes, such as wheel motors control, localization, path planning, graphical view, and so on. Nodes are combined together into a graph and communicate with one another using streaming topics. The use of nodes in ROS provides several benefits to the overall system [44]. Nodes architecture provides benefits: additional fault tolerance, as crashes are isolated to individual nodes and reduced code complexity in comparison to monolithic systems;

- *Messages*: ROS nodes communicate with each other with messages: they are a simple data structure comprising typed fields. Standard primitive types (integer, floating-point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs) [45];

- *Topics*: messages are routed via a transport system with publish-subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption [45];

- *Services*: the publish-subscribe model is a very flexible communication paradigm, but its "one-way" transport is not appropriate for request-reply interactions, which are often required in a distributed system. Request-reply is done via services, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name, and a client uses the service by sending the request message and awaiting the reply [45];

- *Master*: The role of the Master is to enable individual ROS nodes to locate one another and to communicate with the peer-to-peer protocol. ROS master provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics. Nodes connect to other nodes directly; the Master only provides lookup information. Nodes that subscribe to a topic will request connections from nodes that publish

that topic and will establish that connection over an agreed upon connection protocol [45].

ROS's core functionality is augmented by a variety of tools that allow developers to visualize and record data, easily navigate the ROS package structures, and create scripts automating complex configuration and setup processes. The addition of these tools greatly increases the capabilities of systems using ROS by simplifying and providing solutions to a number of common robotics development problems. These tools are provided in packages like any other algorithm, but rather than providing implementations of hardware drivers or algorithms for various robotic tasks, these packages provide task and robot-agnostic tools [46]. Following, a brief description of those tools that will make the explanation of this project clearer.

- *rviz*: it is a three-dimensional visualiser used to visualise robots, the environments they work in, and sensor data. It is a highly configurable tool with many different types of visualisations and plugins;

- *rosbag*: it is a command-line tool used to record and playback ROS message data. It uses a file format called bags, which log ROS messages by listening to topics and recording messages as they come in. Playing messages back from a bag is largely the same as having the original nodes that produced the data in the ROS computation graph, making bags a useful tool for recording data used in later development;

- *catkin*: catkin is the ROS build system, it is based on CMake, and is similarly cross-platform, open-source, and language-independent;

- *roslaunch*: it is a tool used to launch multiple ROS nodes both locally and remotely and set parameters on the ROS parameter server. Roslaunch configuration files written using XML can easily automate a complex startup and configuration process into a single command. Roslaunch scripts can include other roslaunch scripts, launch nodes on specific machines, and even restart processes that die during execution.

ROS contains many open-source implementations of common robotics functionality and algorithms. These open-source implementations are organized into packages. Many packages are included as part of ROS distributions, while others may be developed by individuals and distributed through code-sharing sites [46]. Among the packages included in ROS there is one worth mentioning called gazebo, which integrates tools to use a simulation environment. ROS has become a standard for robot programming over the last ten years for its flexible characteristics and focus on collaborative robotics software development. It is also very interesting for autonomous driving software development. The open-source collections of libraries

and tools, together with its ecosystem and community, are at the base of the success of ROS as a university-level autonomous driving framework, and for the same reasons, it has been chosen for the deployment of the autonomous system under investigation.

In the following subsections, all the nodes belonging to the whole perception pipeline will be analysed, focusing on their tasks and which ROS messages types are taken into account.

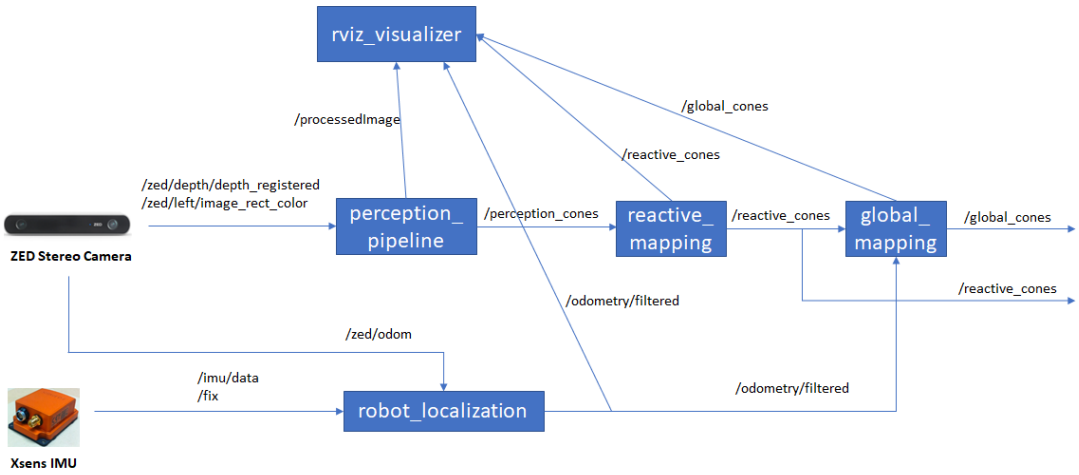


Figure 3.13: Simplified perception pipeline

3.3.2 Perception

As shown in Figure 3.12, perception is the first task to accomplish. In fact, to build the racetrack map, it is necessary to detect the traffic cones that delimits it. The proposed perception task is fully developed by means of a properly designed Python and C++ node in ROS named **perception_pipeline**. This node aims to perform the visual perception process understanding the cone colour and performing how far the cones are from the camera. As already explained, the investigated method does not involve the LiDAR and does not implement a sensor fusion technique between stereo camera and LiDAR since it is intended to build a local map from the sensors even in case of a failure of one of the two sensors. The simplified pipeline used is based on the GitHub PerceptionAndSlam_KTHSFSD1718 [47] done by KTH Formula Student Driverless showing in Figure 3.13.

The proposed vision-based perception pipeline is performed thanks to the Python node already defined, which subscribes to the two stereo camera images: the image colour and the depth image. All the computations and passages that the node does will be explained deeply in Chapter 4. However, this node publishes two

topics: the first one is merely for the visualisation purpose by means to use it to see in the rviz visualiser which cones are detected and how far away they are (**/processedImage**). Instead, the second one is the local map that consists in a NumPy array composed by four columns where the first three represent the local coordinates of the cones detected, while the last column shows the class, i.e. the colours, of the cones (**/perception_cones**). This last topic is fundamental for the whole pipeline since it is feed forward through the reactive mapping node as shown in Figure 3.13. Finally, in Table 3.1 are depicted all the topics needed for the perception pipeline task.

ROS topic	type	Message Content
/left/image_rect_color	Subscriber	Left camera color rectified image
/depth/depth_registered	Subscriber	Depth map registered on left image
/processedImage	Publisher	Bounding boxes and depth visualizer
/perception_cones	Publisher	NumPy array composed by 4 columns

Table 3.1: ROS topics involved in Perception

3.3.3 Localization

As already analyzed, the SLAM process is composed of two simultaneous actions: *localization* and *mapping*. Localization is defined as the ability of a robot to establish its position and orientation within a fixed reference frame. In particular, to reach the localization target, it is necessary to solve an estimation problem that consists of fusing the raw data coming from the environmental sensors (such as stereo camera and INS) to obtain robust odometry. However, the race car is not a linear system, so it is needed to estimate the maximum likelihood of random variables over time, like the car velocity. Multi-sensor data fusion covers a fundamental role in designing robust perception pipelines for autonomous driving, in such scenarios where are required to deliver consistent and reliable data even in case of sensor failures or excessive fluctuations of their measurements. The most famous techniques used for sensor fusion tasks are derived from the concept of state estimation, exploiting a probabilistic approach in the usage of statistical inference from observations by different sources [48]. Over the years, multiple methods have been considered for navigation purposes. However, each of them has advantages or drawbacks depending on which operating conditions they work in. For example, the dead-reckoning approach is beneficial for short-distance assessment of positions and vehicle's states; it suffers from error accumulation over time due to numerical integration. On the opposite, absolute measurements are good on a long-distance perspective for assessing trajectories and paths but with small resolution and are

strongly dependent on environmental conditions. Many sensor fusion techniques have been studied and introduced over the years to deliver the optimal estimate of positions and states of vehicles to overcome these drawbacks. Sensor fusion merges numerical data from multiple sensors to achieve an information gain relative to using each sensor individually. In addition, sensor data are combined so that the resulting measurement is less uncertain than those associated with the single readings alone. The estimation problem is solved in the localization field by adopting filtering or smoothing techniques, starting from raw sensor measurements. The most common filtering technique is the Kalman Filter which R. E. Kalman introduced in 1960. The Kalman filter is an optimal recursive data processing algorithm [49] that uses a series of measurements observed over time, corrupted by statistical noise and other inaccuracies. It produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone [50]. This is possible by using some basic knowledge of the system and the measurement device, the statistical description of the system noise, measurement error, uncertainty in the dynamics, and any available information about the initial conditions of the variables of interest. The Kalman filter algorithm is recursive and leverages the estimation in real-time of the joint probability distribution over the variables for each time-frame to deliver the optimal estimate.

The basic principle of the Kalman filters is presented in Figure 3.14. However, the theory behind the Kalman filters is out of this thesis scope.

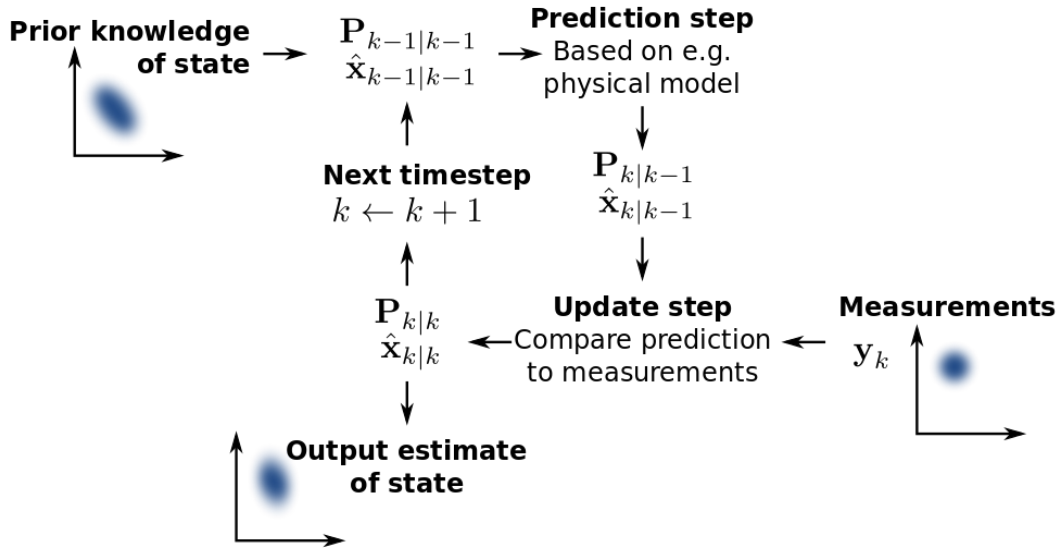


Figure 3.14: Basic principle of Kalman filters

Although the Kalman filter is a good estimator, its working principle is only for a linear system. The proposed pipeline mainly focuses on the non-linear Kalman

filtering techniques that extend the operational principles toward the non-linear systems. Estimation of non-linear systems is fundamental in implementing Kalman theory because almost all practical methods involve non-linearities of some kind. In a large portion of the field of application of state estimation algorithms, the systems considered are non-linear; thus, the extension of the Kalman theory to this class of systems was straightforward. These filters are also justified by some studies that have tried to develop many other approximations. Still, unfortunately, most of them are computationally too demanding or require specific assumptions on the form of the process that are not satisfied in practice by any physical system. Therefore, Kalman filters remain the most widely used estimation algorithm. The most common application of Kalman theory to non-linear systems is represented by the well-known Extended Kalman Filter (EKF). In fact, in the EKF the state transition and observation models do not need to be linear functions of the state but must be differentiable functions. The central operation performed by the EKF is the propagation of the Gaussian random variable (GRV) that are the only input accepted by the filter through the system dynamics, considering a first-order linearization. The problem with this kind of propagation through non-linear system dynamics is that the operation can introduce huge errors in the true posterior mean and covariance of the transformed GRV. This may lead to suboptimal performance and sometimes cause the divergence of the filter. In particular, those drawbacks are directly linked in the case of localization, where GNSS measurements introduce high non-linearities of the system's dynamics, and thus cause the EKF to sub-optimal performances. Although EKF maintains the elegant and computational efficiency of the Kalman filter, its limitations moved toward the investigation of alternative methods able to propagate first-order linearization through non-linear dynamics. To overcome the EKF limitations, the problem has been addressed using a deterministic sampling approach [51]. The unscented transformation (UT) was developed for propagating mean and covariance information through the non-linear transformations. This method has been proven to be more accurate, easier to implement, and uses the same order of calculations as a linearization of the same system. The result is the so-called Unscented Kalman filter (UKF). In the UKF, the state distribution is again approximated by a GRV, but is now represented using a minimal set of carefully chosen sample points, called sigma-points. These sample points completely capture the true mean and covariance of the GRV, and when propagated through the true non-linear system, the filter can capture the posterior mean and covariance accurately to the 3rd order Taylor series expansion for any non-linearity. To put it into perspective, the EKF only achieves first-order accuracy, and remarkably, the computational complexity of the UKF remain in the same order as that of the EKF [52][53]. The localization pipeline is fully developed by means of properly designed Python and C++ nodes in ROS. As shown in Figure 3.15, the main nodes composing the localization stack are the **sbg_driver**, the

odometry_publisher node, and the **robot_localization** package.

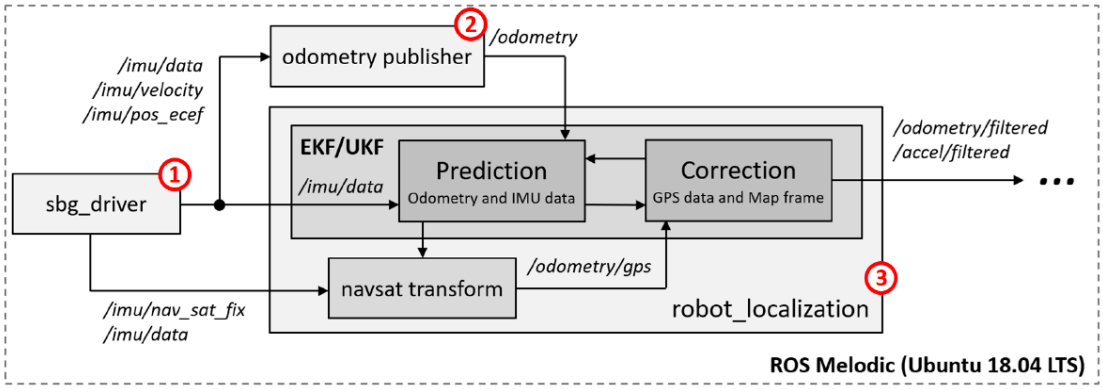


Figure 3.15: Custom ROS packages block diagram representation with topics [54]

1. **sbg_driver node**
   The first node is the SBG Systems proprietary C++ node [55]. It publishes the filtered sensor data at a custom rate decided for each signal based on the required robustness. The standard output rate for the Ellipse-N sensor is 200 Hz for each signal. This node publishes proprietary ROS messages shown in Table 3.2. In particular, all the topics are referred to the main reference frame of the IMU and are reported in the Earth-Centered-Earth-Fixed (ECEF) world frame. SBG Systems propose this node as a ready-to-use package, and it is mainly used for INS initialization.

ROS topic	type	Message Content
/imu/data	Publisher	IMU measurements from INS
/imu/nav_sat_fix	Publisher	Raw GPS data from GNSS
/imu/pos_ecef	Publisher	ECEF position from INS
/imu/velocity	Publisher	Linear/angular twist from INS

Table 3.2: ROS topics involved in sbg_driver node

2. **odometry_publisher node**
   This node has been designed to properly arrange the information coming from the sbg_driver to be used by the filtering node. The odometry_publisher oversees the filling of a standard ROS topic called /odometry. It consists of two geometry messages: */pose* representing the position and orientation of the robot, while */twist* contains the information about linear and angular velocities.

To sum up, the resulting odometry message is called */odometry*, and it reports the position, orientation, and speed of the robot in the East-North-Up (ENU) frame. Moreover, the node performs the transformation between the ECEF frame into the ENU frame. The theory behind the transformations between the two reference frames is behind this project. A detailed explanation of it is reported in [56].

3. **robot_localization Package**
   The robot_localization package covers the last part of the localization pipeline. It is a standard ROS collection of nodes written in C++ freely available among ROS.org libraries [57][58], and it represents the central software component of the localization method. It embeds both the EKF and UKF algorithms that can be appropriately adapted to the application's needs to run separate instances of sensor fusion. The node accepts standard ROS topics as input data for estimation and publishes the transforms between map, odometry and base link frame of the system, together with the filtered states. The two implementations of EKF and UKF, chosen accordingly to the wanted estimator, are used to fuse the data coming from the INS solution with the raw data of the GNSS, adding the information on the frames dependencies. Alongside the state estimation algorithms, a navsat_transform node integrates the GNSS raw data in the measurements and is responsible for the global positioning information update. In addition, the odometry data comes from the odometry_publisher node, which provides the state estimation instances with a complete odometry message called /odometry. The state estimators are also supplied with the IMU data message called /imu/data, coming from the INS via the sbg_driver node as described before. The local instance of the state estimator leverages /odometry and /imu/data to establish the transform between the odom and base link frame and to publish the filtered odometry message under the name of */odometry/filtered*. In the meantime, the internal navsat transform node acquires the GNSS raw data from the INS through the topic /imu/pos ecef of the sbg driver and the heading information from the /imu/data, providing a new odometry message with the global positioning information called /odometry/filtered_map.

   As a result, an **/odometry/filtered** topic containing position and orientation information and a **/tf** message containing the transformations among the reference frames are obtained [59]. The frames considered by the package are the *base_frame*, the *odom* frame and the *map* frame, which are standard reference frames used in ROS environment for mobile robots localization and mapping.

System Integration

Figure 3.16 represents the overview of the localization pipeline designed and

integrated into the complete architecture of the autonomous system from a software/hardware point of view. This figure makes it possible to see another GNSS receiver: EMLID Reach M+ RTK GNSS module [60]. Its benefits is under investigation trying to fit it in the whole pipeline to obtain a redundancy using two GNSSs. Furthermore, the proposed method is again composed of standard messages in ROS environment, commonly used by different pre-defined packages for running mapping or planning algorithms. One of the main design targets achieved by this localization pipeline is to provide the mapping algorithm and the path planner with a precise and robust odometry message complete of means and covariances. The/accel/filtered topic also provides other relevant states of the vehicle while the positioning information for post-processing can be found in the /gps/filtered topic.

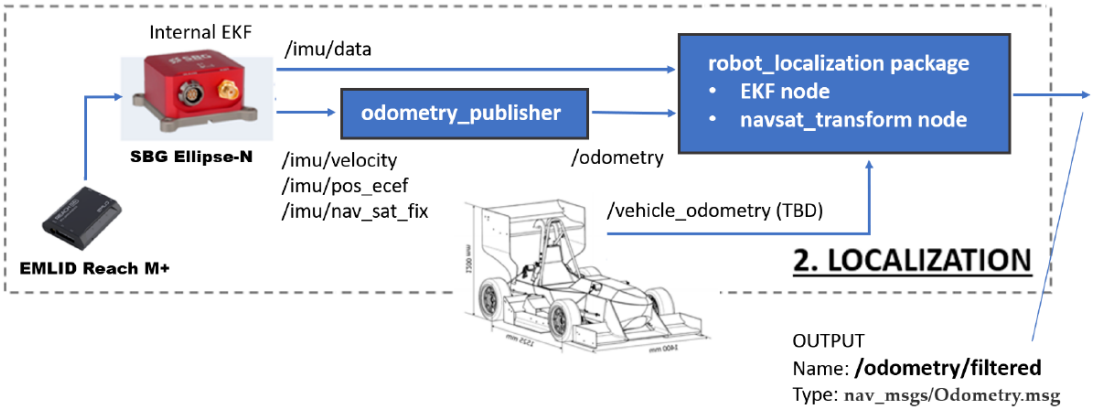


Figure 3.16: Software and Hardware architecture for Localization

3.3.4 Mapping

As explained previously, the mapping is a characteristic process of the SLAM where the cones sensed by the vision sensors are then plotted on a map representing the whole lap of the racetrack. In general, SLAM algorithms are well documented solutions in literature and have been robust and reliable for state estimation and racing applications [61]. However, the main drawback of the SLAM approach when dealing with unknown racetracks or urban maps is the need for a software stack that is able to run a global mapping routine during a first low-pace lap of the course to achieve a good level of details of the map. Then, the vehicle can navigate for the remaining laps in a perfectly known environment in a pure autonomous roaming condition. This process is poorly applicable to commercial vehicles in urban areas since accurate digital mapping is one of the main concerns of autonomous technology providers. However, the limitation of the approach is the impossibility of tackling an unknown environment for the first time with a high level of confidence. Moreover, in the autonomous racing scenario, it means sacrificing the vehicle's dynamic performance during the initial lap while building the environment for the global path planner. This process is fundamental for car navigation, thus, for path-planning and actuation. In particular, during the first lap of the race, the racetrack map is unknown. In this phase, the trajectory planning needs to be short-term or reactive to the cones detected immediately. Once the first lap is completed, the racetrack map is built. Thus, trajectory planning becomes long term.

The mapping process has been implemented through two ROS nodes written in Python named *reactive mapping* and *global mapping*.

reactive_mapping node
The reactive mapping ROS node is fundamental in the unknown racetrack's first lap, devoting to the path planning task. It takes as input the /perception_cones topic and produces as output the /reactive_cones topic. To sum up, the topics needed are shown in Table 3.3.

ROS topic	type	Message Content
/perception_cones	Subscriber	NumPy array composed by 4 columns
/reactive_cones	Publisher	NumPy array composed by 4 columns

Table 3.3: ROS topics involved in reactive_mapping node

Once again, the */perception_cones* topic is an extended one-dimensional array produced by the perception_pipeline ROS node with a dimension of 4N. In particular, N represents the number of cones depicted at every single frame, while 4 represents the parameters needed to identify each cone. They are in a NumPy

array of the form $[x, y, z, colour]$. The first three values are the three-dimensional coordinates of the detected cone in the camera reference frame, whereas colour represents the numeric number linked with the class of the cone (i.e. 0 yellow, 1 blue and 2 orange). The node publishes the */reactive_cones* topic with an extended one-dimensional array formed by a size of multiples of 4. However, for each cone detected there are 4 values with the form $[centroid\_x, centroid\_y, variance, colour]$, where the first two are centroid of the cone positions, the third value represents the position variance, and finally, the last one represents the colour of the cone.

Figure 3.17: Reactive mapping node

Briefly, the reactive mapping node is composed of two functions named the *callback_perception()* and the *cluster_frames()* respectively. An example can be used to justify its working principle to make everything more straightforward. If the stereo camera detects N cones in every frame, the *callback_perception()* method integrates data from three successive frames and creates a 3N×4 matrix that will be the input of the *cluster_frames()* function which is called within the *callback_perception()* one. The *cluster_frames()* method assigns a cluster identification number to every cone. Suppose the number of cones in a cluster is major than the 65% of the maximum number of times that a cone can be detected (each cone can be detected maximum 3 times because three frames at a time are analyzed, so each cone should be seen at least 2 times). In that case, they are legitimated, and their positions are used to compute centroid (the mean value) and the variance in cone positions. In the end, the */reactive_cones* topic is published.

Global mapping node

The global mapping node is a Python ROS node that aims to create a global map of the racetrack as accurately as possible since all the long-term control decisions of the car are based on this map. The origin of the global map created by this ROS node overlaps with the initial position of the car. This ROS node receives as inputs two topic: the */odometry/filtered* topic coming from the localization package containing the position and the orientation of the vehicle with respect to its initial state and the */reactive_cones* which is the topic publishes by the reactive mapping node. Thus, if some cones in the global map should not be there or they are not

stored in the map, the car could have some problems such as stopping prematurely, go toward a not real cone or in the worst-case crashes against a cone that is not marked in the map. Furthermore, the global mapping output is constituted by the */global_map_markers* topic which is an extended one-dimensional vector with the following format $[x, y, colour, covariance, hits, inFOV, id]$ where: x and y are the position coordinates in the global reference frame, *colour* is the cone's color code, *covariance* is the covariance related to the position, *hits* is the number of time that a cone has been detected, *inFOV* is a flag that indicates if a cone is in the camera field of view and *id* is an identification number that characterized each cone.

ROS topic	type	Message Content
/reactive_cones	Subscriber	NumPy array composed by 4 columns
/odometry/filtered	Subscriber	Vehicle position and orientation
/global_map_markers	Publisher	NumPy array composed by 7 columns

Table 3.4: ROS topics involved in global_mapping node

This node is characterized by three main functions: the *callback_odom()*, the *callback_reactive()* and the *update_cone_db()*. The first method (callback_odom) computes from the */odometry/filtered* topic the translation and the rotation matrices that need to pass from the reactive reference frame to the global one. The second function (callback_reactive) needs to take from the */reactive_cones* topic the pose and the colour of each cone detected by the camera expressed in the reactive reference frame. And lastly, the third method is the update_cone_db(). This function is called with a fixed frequency called update frequency that is a design parameter. The logic behind this method is that if a certain cone is seen more times during the travelling along the circuit, the covariance related to its position decreases, on the other hand, if a car, performing more laps of the same racetrack, does not see a certain cone frequently, the cone's position covariance increase till the covariance reaches a minimum value and the cone is removed from the map [47].

Figure 3.18: Global mapping node

Chapter 4

Package Development

This chapter deals with the final implementation of the perception pipeline mounted on the vehicle exploring the real implementation of the two neural networks identified. The theoretical background presented so far, together with the investigated hardware and software configurations, is the basis of the decision-making process to reach the final thesis scope. With a brief introduction about the perception pipeline node already explained in Section 3.3.2, the first part of this chapter is devoted to justify and analyze which convolutional neural networks are depicted, mainly focusing to the real implementation of the stereocamera-based perception algorithm. It is worth noticing that this pipeline might be redundant to the LiDAR-based Perception algorithm, increasing the robustness of the measurements of the cone's positions. Nevertheless, it performs a peculiar task since it can estimate not only the position of the detected track boundaries but also the colour of the detected cones in front of the sensors. Then, the two methods such as Single Shot Detector (SSD) and You Only Look Once version Four (YOLOv4) design architectures together with the matching method between the coloured image and depth image for distance estimation are presented. Finally, the rapid platform implemented using a 1/10 radio-controlled vehicle is explained.

4.1 Stereocamera-based Perception Algorithm

In Section 3.1.1 all the main features of the ZED stereo camera have been highlighted, specifying which role the camera covers in this project. Here, the camera configuration is presented starting from the first step done: the **Camera Calibration**. In Chapter 3, it was analyzed how a stereo camera might be calibrated using a chessboard. In our case, the ZED stereolabs company already provides a *ZED Calibration tool* that aims to calibrate and give the main parameters about the intrinsic camera ones. Those parameters cover an important role in the perception

field since moving from the pixel reference frame to the world reference frame a transformation is needed throughout those camera parameters. The calibration phase was done in a dark room at LIM laboratory using the computer and the stereo camera. The tool asks to point the camera toward the screen where a chessboard appears. With precise camera movements, the camera is calibrated, giving different values with respect to the video mode chosen. The video mode has been investigated to satisfy the trade-off among a satisfactory frame rate (FPS) and the energy resources deploying the camera on NVIDIA Jetson AGX XAVIER, which means the minimum output resolution to allow good cone detections. Firstly, it is needed to say that to increase the robustness of the cone detection a LiDAR-based Perception algorithm will be developed from the Perception division of the Squadra Corse PoliTo Driverless, in Chapter 6 more insights will be explained. Moreover, each of the two pipelines is redundant with respect to the other one in order to prevent inaccuracies in the obstacle detection process. Recall that the LiDAR used is a Velodyne VLP-16 that provides a full 360° view of the surrounding environment at 10 Hz to obtain an accurate real-time 3D data reconstruction recorded by the 16 light channels. Starting from the LiDAR frequency, the video mode chosen to fulfil this trade-off is the 720p with an output resolution of 2560x720 side by side, and a frame rate of 30 FPS, as it is possible to see from Figure 3.6. As a matter of fact, the proposed vision-based algorithm can detect obstacles in the input images and draw the bounding boxes at a frequency up to 30 Hz in the actual real-time application. Specifically, the left camera of the ZED is used for the cone recognition through the convolutional neural networks, while both the images are used to compute the depth map using triangulation from the geometric model of non-distorted rectified up to 10 m [62].

After this premises the intrinsic camera parameters are found straightforward looking at the file where the camera parameters are saved, specifically, the one with visual mode 720p and the left camera of the ZED stereo camera. Those values are shown in table 4.1:

parameters	**values**
focal length fx	699.401
focal length fy	699.401
principal point cx	611.641
principal point cy	357.481

Table 4.1: Intrinsic camera parameters from Calibration stage: left cam HD

The other parameters that should be decided from the stereo camera point of view are:

- Sensing mode: **Standard mode**

- Depth mode: **Performance**
- Depth range: **from 0.3 m to 20 m**

These configuration parameters can be modified in two files belonging to the *zed_wrapper* node that are *param/common.yaml* (Figure A.1) and *param/zed.yaml* (Figure A.2) shown in Appendix A.

The stereocamera-based perception algorithm has not been implemented from scratch but adapting and optimizing the one from [47]. Thus, the **perception_pipeline node** developed by KTH FSD is a Python ROS node that takes as inputs (subscribe) two images coming from the ZED stereo camera: the coloured image and the depth image, and produces as output (publish) the /perception_cones topic (Figure 4.1).

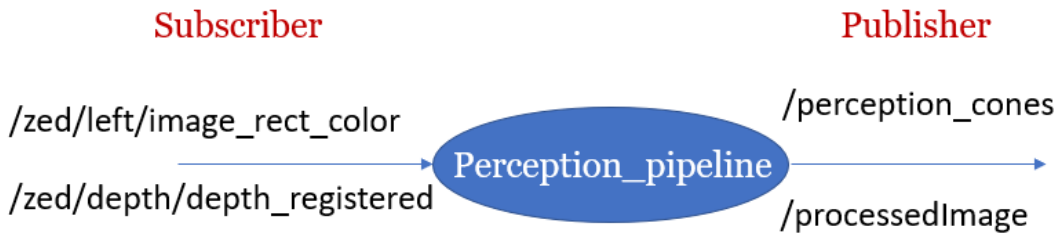


Figure 4.1: Perception ROS node scheme

Both the Single Shot Detector and the You Only Look Once algorithms are developed into this node, having some aspects in common and others not. In particular, they shared almost perfectly the structure of the "main" and some callback functions needed for the image acquisitions. In general, the obstacle distance with respect to the sensor is computed by matching the detected bounding boxes representing the obstacles with the recorded depth map from the ZED stereo camera. Moreover, the camera can compute depth map using triangulation from the geometric model of non-distorted rectified cameras. In general, the depth D of each point p is computed as:

$$D = \frac{fb}{xi^L - xi^R} \tag{4.1}$$

Where:

- f: focal length
- b: baseline distance of the stereo camera
- $xi^L - xi^R$: disparity value

The focal length f is assumed equal for the two cameras retaining that they are co-planar with parallel optical axes. The left camera is assumed as the origin frame for the resulting depth map. Since high disparity means that the point is close to the stereo camera, it is immediate to say that the disparity map is inversely proportional to the two-dimensional depth map. Moreover, the 3D reconstruction phase uses the depth information in the disparity map along with camera calibration parameters by matching RGB pixels with the two-dimensional coordinates related to the disparity map created with respect to the optical center of the left camera. The result of this process is a dense map of RGB points in 3D coordinates which is accurately obtained for distances lower than 10 m. The obtained 3D reconstructed point-cloud can be finally exploited for estimating the distance of the objects corresponding to the identified bounding boxes. This task is commonly performed by computing the center point in each of the bounding boxes and projecting it into the disparity map. A two-dimensional local map can be computed by using the knowledge of both the position and distance of the detected obstacles. A general block scheme able to describe the starting point of the proposed stereocamera-based perception algorithm is presented in Figure 4.2. The matching stage proposed in this figure is performed in a straightforward way. Once the cone bounding box is generated by one of the two CNNs, the algorithm computes the center value of the bounding box and subsequently takes this point in the depth image obtaining the detected objects with distance. Finally, in the following subsections will be explained deeply all the algorithms highlighting the part in common and the differences.

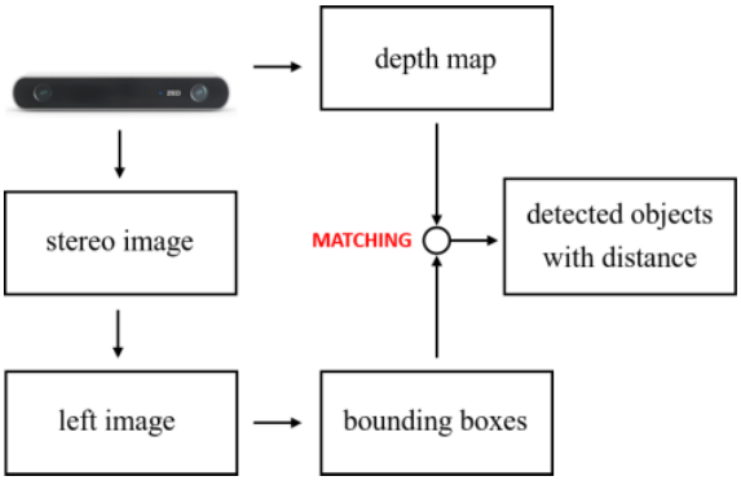


Figure 4.2: Block-scheme of the proposed stereocamera-based perception algorithms

4.1.1 SSD

The first proposed stereocamera-based perception algorithm used in this thesis is the Single Shot Detector algorithm with MobileNetv1 as backbone. The theory behind that algorithm was deeply discussed in Chapter 2 where it was proven that among the one-stage and two-stages object recognition the first ones are able to guarantee a real-time application for autonomous driving. The perception_pipeline node subscribes to image and depth information published by the zed camera. The node passes every image through a convolutional neural network with SSD-mobilenet architecture. The CNN is trained to detect traffic cones in an image and promptly returns a list of bounding boxes where they are present. Thereafter, the node infers the color and calculates the depth of every traffic cone candidate. It publishes an array of detected cones in camera frame. For this purpose, everything happens inside the perception_pipeline node without recalling any other node. This node is composed by 4 elements:

1. perception_pipeline.py
2. Cone_img_processing2.py
3. label_map.pbtxt
4. frozen_cone_graph.pb

The cone recognition is done using *Tensorflow*, it is a free and open-source software library for machine learning and artificial intelligence developed by Google. It can be used across a range of tasks but it has a particular focus on training deep neural networks. Not only Tensorflow, but also *Protobuf* model is needed since it contains everything for a model execution. In particular, the *.pbtxt* file is where the trained model is saved allowing to directly load it in the memory without further training. In the proposed method, only the cones should be identified, so the file label_map.pbtxt contains just one classification word named "cones". The lines here do a match between the category label to the class name, so when the network gives $output = 1$ means that a cone is identified.

The perception_pipeline.py file is the most relevant one. It is composed by the *main* (Figure 4.3) and two *callback functions*. The main is the core of the algorithm where all the computations are performed. It is composed by a first initialization phase which the intrinsic camera parameters, the publishers and subscribers needed are developed. Then, the SSD model is uploaded in the memory with the already trained weights. Subsequently, the last Frame is copied and if the flags belonging to the presence of the coloured and depth image are set to true the image is expanded (just for a TensorFlow convention) and the SSD is feeded. For each cone detected at every frame (indicated with i in the flowchart), it is verified that the detection's

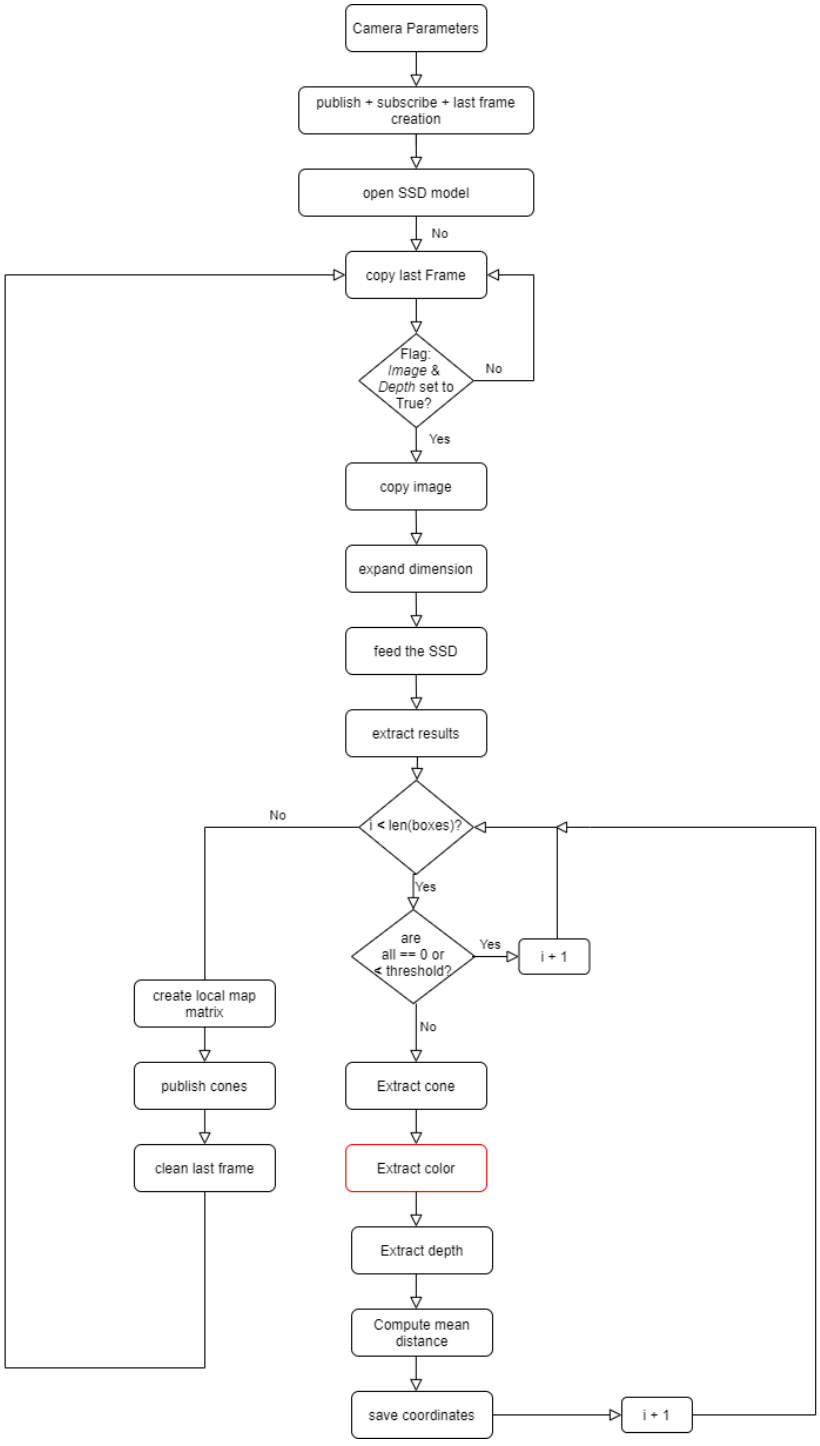
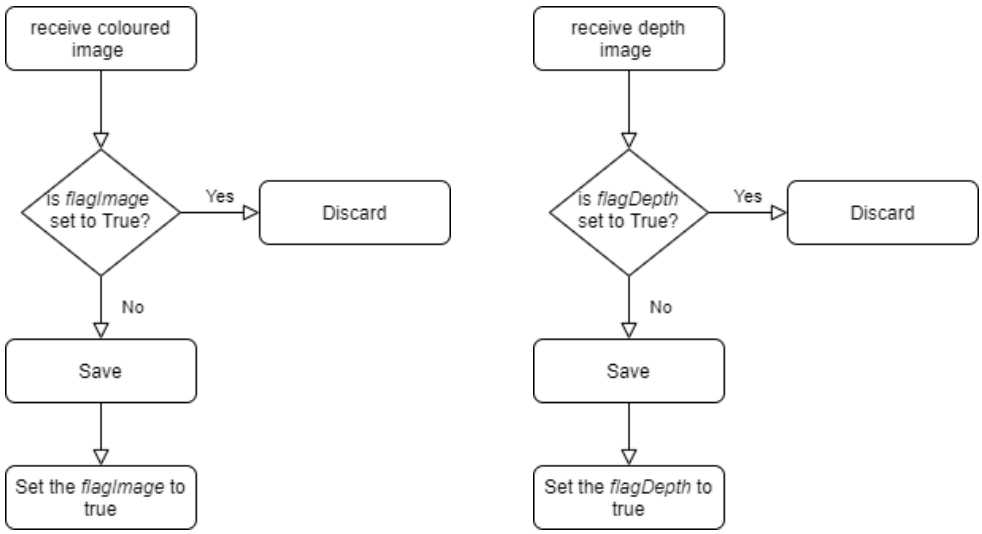


Figure 4.3: main of the SSD algorithm

probability given by the SSD model is above the threshold value of 0.5. The colour extraction (red box in Figure 4.3) is performed recalling a function defined in the script cone_img_processing.py, where after a first transformation from pixel to HSV (Hue Saturation Brightness) the pixels are counted for every colour. Then, the max colour is extracted leading towards the colour decision.

Finally, the depth computation is performed computing the mean distance, and after that, save the coordinates and the colour class in /perception_cones topic. As already explained in Section 3.3.2 the /perception_cones is a 4N matrix, where N is the number of the cones detected in in each Frame. Every cone is characterized by a data packet of 4 values whose format is $[x, y, z, colour]$ where x, y, z are the three-dimensional coordinates of detected cone in the stereo camera reference frame and colour is the numeric code that identifies the detected cone's colour.

The two callback functions are fundamental in the activation of the flags. In this way the asynchronous programming is fulfilled guaranteeing that both the images (coloured and depth ones) are arrived together. Figure 4.4 shows the flowchart of the two callback functions.



(a) callback storage image **(b)** callback depth

Figure 4.4: Callback functions.

4.1.2 Yolov4

As presented in Chapter 2, the Yolo has a better performance than the SSD algorithm. Therefore, its implementation on the previous perception pipeline architecture will be presented.

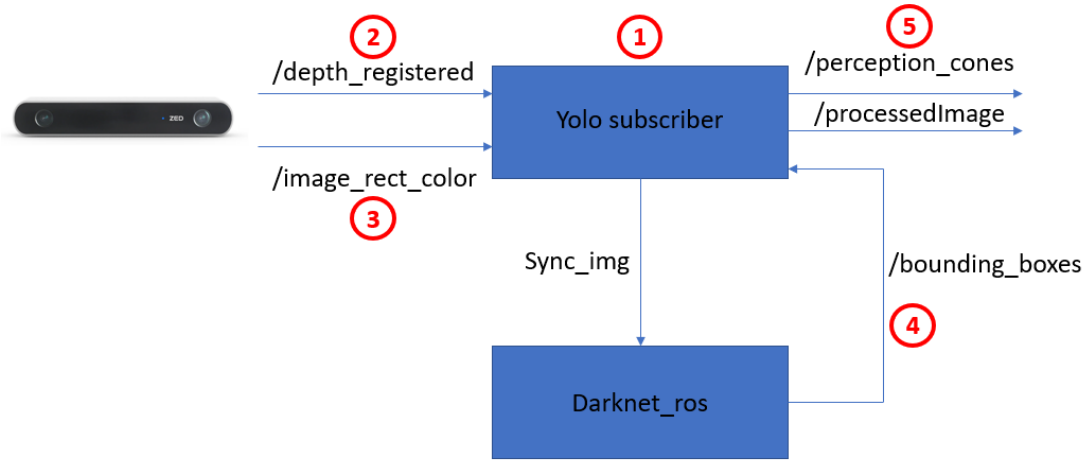


Figure 4.5: Yolo block diagram

The block diagram for implementing the cone recognition and distance estimation is shown in Figure 4.5. In this case two nodes are used: the already presented **perception_pipeline** node and the **darknet_ros** node. Inside the perception_pipeline node, the yolo_subscriber.py script covers the main's role, previously covered by the perception_pipeline.py in the SSD. These two mains have an almost identical structure just with a small variation of the logic behind the depth computation. Figure 4.6 represents the *main* regarding the Yolo algorithm in which it represented by the red number 1 in Figure 4.5.

Also the Yolo callback functions have similar behaviour to the SSD ones. In fact, their most important task is to allow a proper behaviour setting to true the flags that are devoted to checking if the images have arrived or not. However, not only the images coming from the stereo camera but also another flag called FlagReady must be set to true telling the system that all the data necessary for processing has come. The flag related to the callback_storage_image (Figure 4.7a) is named *flagImage* and it is set to true when the coloured image has arrived. Moreover, this callback sends the image to Yolo allowing the cone recognition to take place. The callback_depth (Figure 4.7b) is devoted for acquiring the depth image from the stereo camera and set the *flagDepth* to true. These two callback functions are represented in Figure 4.5 by red number 2 for the callback_depth and by the red number 3 for the callback_storage_image.

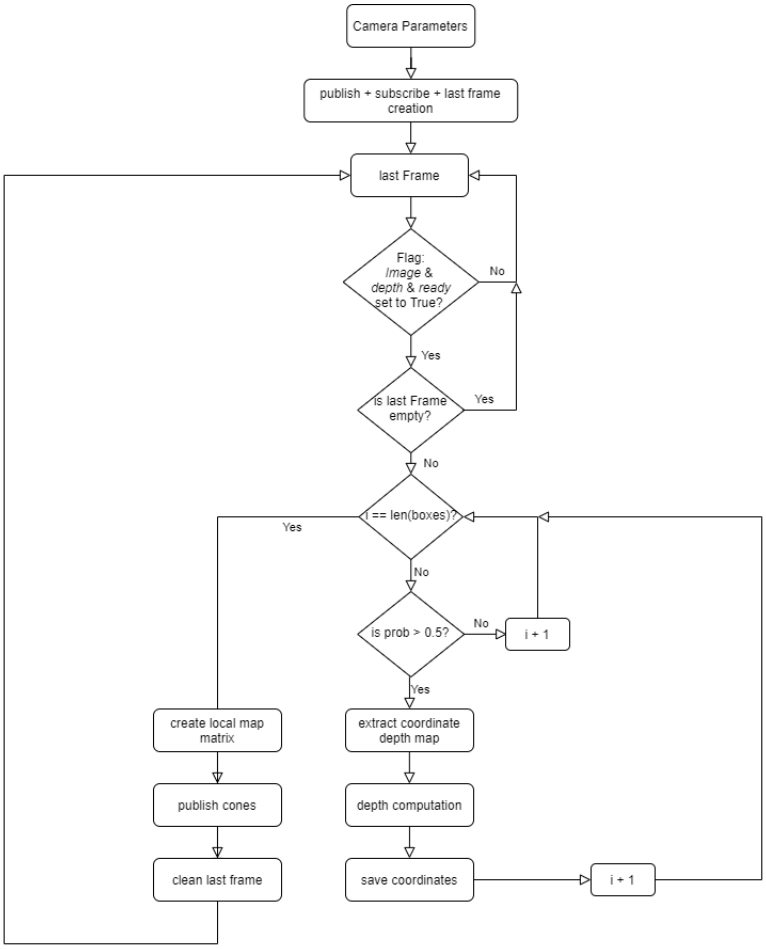
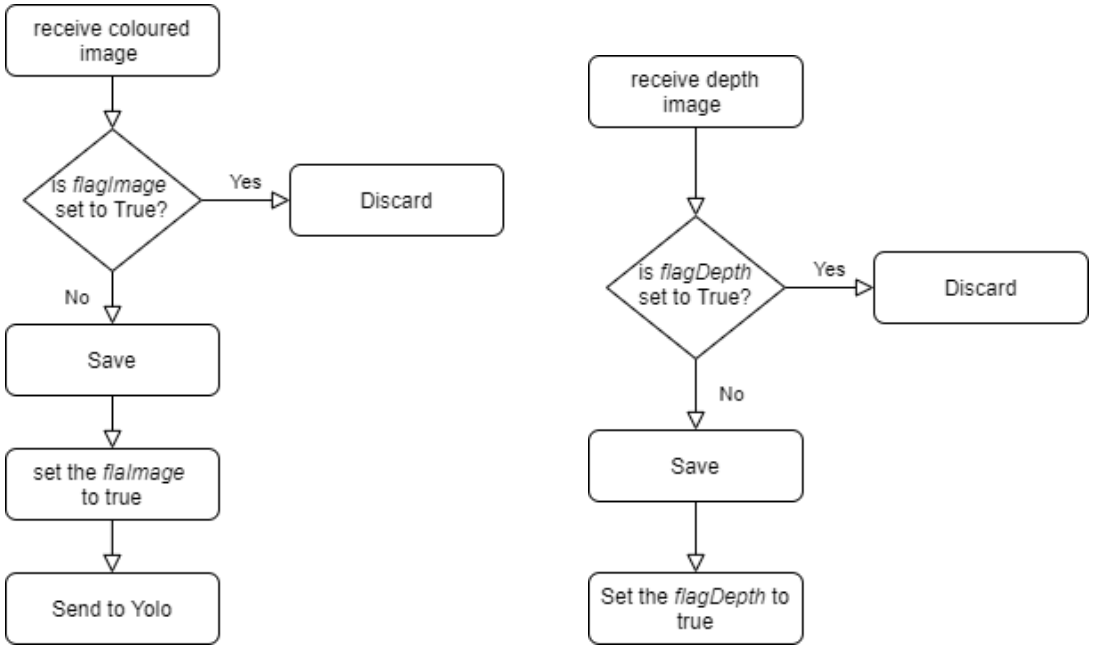


Figure 4.6: main of the Yolo algorithm inside the yolo_subscriber.py

As already explained, Yolo algorithm is performed by using two nodes which need to be synchronized. For this purpose, before processing the depth computation, a fictitious image is sent from the yolo_subscriber to the darknet_ros node. This image, called *sync_img*, is completely equal to the image_rect_color coming from the stereo camera. The perception_pipeline node publishes it, and the Darknet_ros subscribes it. Yolo is developed over Darknet that is an open source neural network framework written in C and CUDA. It is fast and supports CPU and GPU computation. The fundamental element for the cone recognition performed by Yolo in the Darknet framework is roslaunch file: *Darknet_ros.launch*. Firstly, it loads the parameter configuration of the neural network and the weights. For this project thesis, the weights used are "tiny"; this means that the neural network has been trained with only hundreds of pictures, which could not allow robust cone recognition for our application. Then, the bounding_boxes topic is published. It is a



(a) callback storage image **(b)** callback depth

Figure 4.7: Callback functions.

costumed message contains $[probability, xmin, ymin, xmax, ymax, id, class]$, where class represents the category that Yolo assigns it to. After the /bounding_boxes topic is published by Darknet, the callback_yolo is used to append the parameters proper of each cone and set to true the last flag named *flagReady*. The flowchart related to this callback is shown in Figure 4.8, while in Figure 4.5 this callback is represented by the red number 4.

Going back in the yolo_subscriber script, all the data needed for the post processing are obtained and all the flags are set to true. To summarize, Table 4.2 represents what it means when the flags are set to true.

Flag name	**Flag objective**
FlagImage	Coloured image coming from the stereo camera has arrived
FlagDepth	Depth image coming from the stereo camera has arrived
FlagReady	The bounding boxes coming from the Darknet node has arrived

Table 4.2: Flags needed to perform the asynchronous programming

When the flags are on, the main script reaches the end performing the match between the bounding boxes and the depth image. Finally, since the */perception_cones* topic is needed to feed the reactive mapping node, the publish_cones

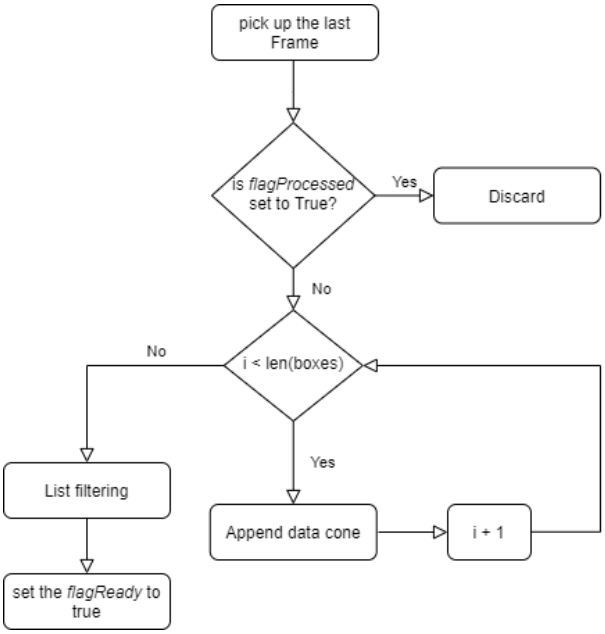


Figure 4.8: Callback yolo

function publishes the topic formed as NumPy array with the cone types and positions (Figure 4.9) represented by the number 5 in Figure 4.5.

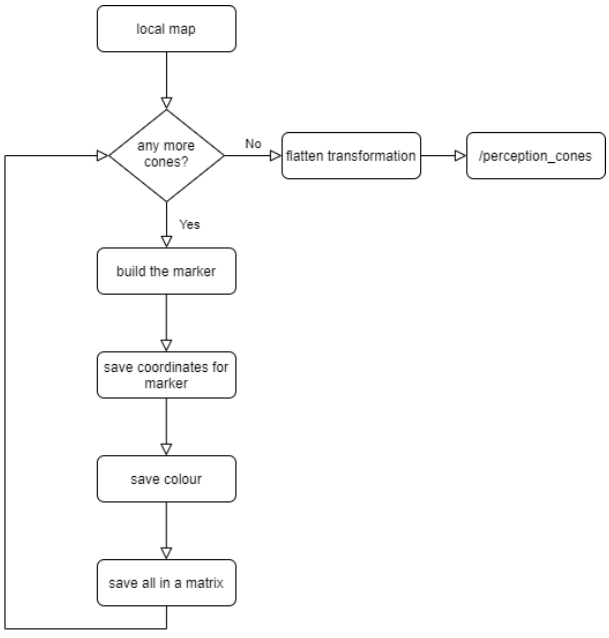


Figure 4.9: Publish cones function

4.2 Rapid Platform with an RC car

Having a racing prototype always ready for multiple testing sessions is very difficult. All the division teams work on the car methodically to transform the vehicle from a human-driven car into an autonomous driving one, for this reason, an hardware platform has been developed. Rapid prototyping is a group of techniques used to quickly fabricate a scale model of a physical part or assembly using three-dimensional computer aided design (CAD) data. So, the rapid platform built in this thesis project aims to create a platform able to test not only the proposed algorithms more quickly with respect to the SC19D car, but also to allow the other divisions of Squadra Corse PoliTo Driverless to work directly on the car and create meanwhile the right supports for the sensors used. The idea was to implement, using a radio controlled vehicle, a platform featured with all the sensors involved in the stereo camera based perception algorithm. A first choice was about the RC vehicle. In general RC car can be classified basing on two main features: the scale of the vehicle and the body shape. Although the RC vehicles are produced with standard scales, they must be big enough to carry all the required hardware components. Initially, the purpose was to make the hardware configuration using the robust plastic shell provided by the RC vehicle, this means at least for the first moment, also the body shape was retained important in the decision phase. The vehicle chosen for this project is a XinleHong 9125 1/10 2.4G 4 Wheel-Driving 46km/h RC Car Short Course Truck capable of reaching speeds up to 46km/h shown in Figure 4.10.

Figure 4.10: 1/10 Radio-Controlled car

While in Table 4.3 are collected its main features:

Mechanical parameters	Car size: 34.5x30.5x16.5 cm Car Weight: 1530g Scale: 1/10
Battery	Type: 7.4V 1600mAh Lipo Using time: about 10 minutes Charger time: about 2.5 hours
Radio transmitter	Frequency: 2.4GHz Control distance: about 80m Battery: 3x AA
Power System	RC Radio receiver and Integrated 60A ESC RC Motor:390 Brushed RC Servo: 2.2Kg

Table 4.3: Radio-controlled components

In Figure 4.11 the hardware components carried on the vehicle are shown:

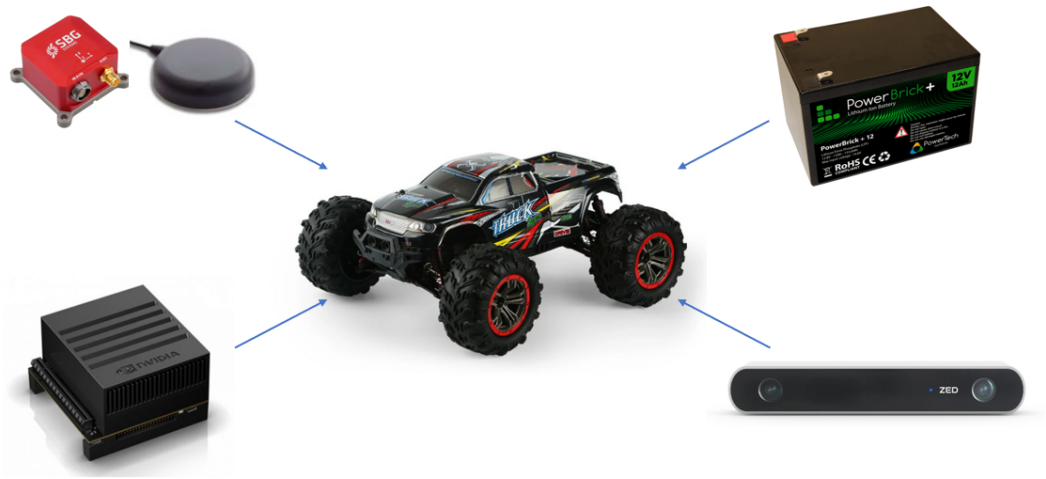


Figure 4.11: Sensors and RC car used for implementing the rapid platform

As already explained in Subsection 3.3.2, the stereocamera-based perception pipeline needs only a stereo camera, the NVIDIA Jetson Xavier computing platform, and a battery to supply it. Nevertheless, this rapid platform aims to also test the localization and mapping tasks, this is why also the INS and the connected GNSS are carry on the vehicle.

Two main radio-controlled vehicle configurations can be identified during this project. The first configuration was done using the shell of the vehicle. Thanks to

the rear flat shape of the vehicle, the Nvidia was positioned there. The camera was set in front of the vehicle while the INS was momentarily put over the DC-DC boost. This converter is fundamental to provide the right voltage supply to the embedded computer used. In fact, the Nvidia datasheet suggests using a converter to increase the 12 V coming from the battery to 19.5 V to allow running at full clock speed. Finally, a 1/16 trailer was used to carry on the battery since due to its physical dimensions there were not ways to install it on the RC car. This proposed set up was used during the first months of the project testing the the correctness of the pipeline. On Figure 4.12, the first configuration is shown:

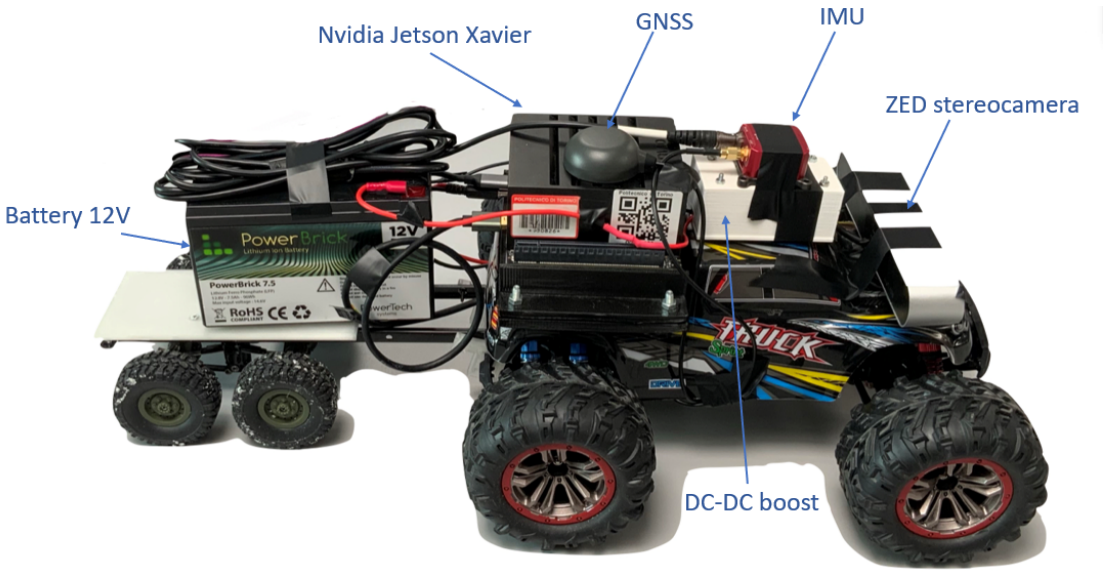


Figure 4.12: First Hardware configuration of the RC car

The second configuration has been developed during the last part of the project. Its peculiarity are the shell removal and the 3D stamps of a platform anchored to the body of the radio-controlled vehicle. The platform was designed using SolidWorks and then printed in LIM laboratory optimizing all the free spaces and guaranteeing a robust wiring connection among all the sensors. Figure 4.13 represents the new configuration that will be used to test the whole perception pipeline except the trailer with the battery that are the same of the previous set up. This configuration is easier to manage with respect to the first one allowing a faster battery change and a rapid checking of the correctness of the connections due to its cleaner architecture.

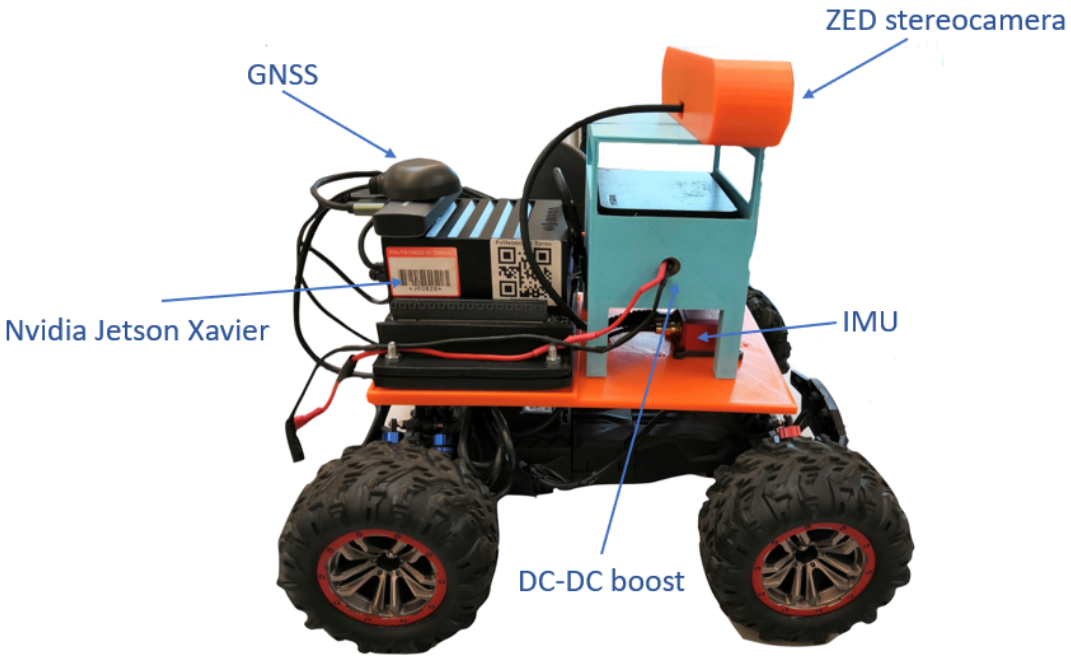
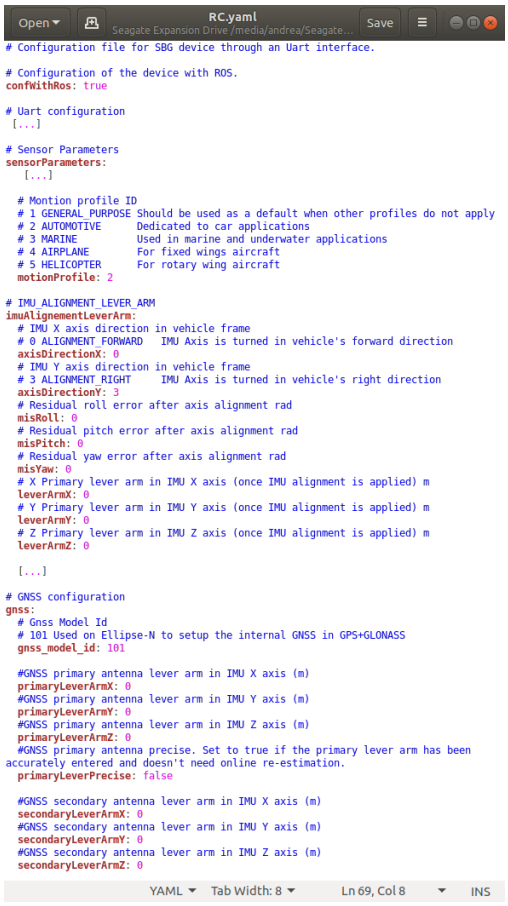


Figure 4.13: Second Hardware configuration of the RC car

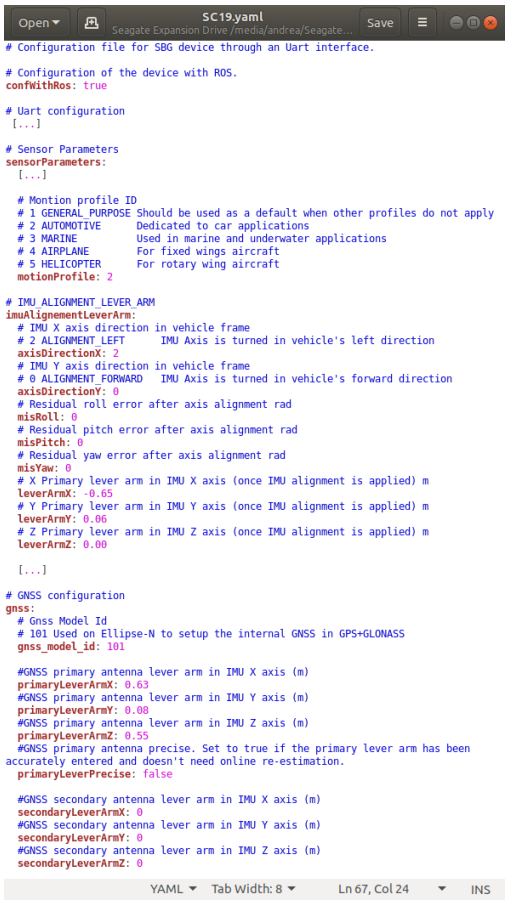
Software setting for the radio-controlled vehicle

The Hardware set up introduced previously shown sensors placed close to each other. Hence, the cables connecting the sensors are reduced with respect to the hardware configuration of the SC19D racing prototype by Squadra Corse PoliTo. These reductions in the RC vehicle size and in the sensors configurations lead to some software modifications in term of intrinsic parameters and so on. For example, relating to the localization stack, one of the crucial operations performed by the INS sensors is the alignment phase. This preliminary stage must be done before the nominal testing conditions which covers an important role to achieve sub-meter accuracy of the estimates. The software setup consists in the initialization of the vehicle's parameters in the sbg_driver node composed by an embedded routine for accounting misalignment and mechanical lever arms. The alignment accounts for the device rough orientation on the vehicle and the misalignment angles, while the main lever arm is the distance of the INS body from the centre of motion of the vehicle. Furthermore, concerning the GNSS antenna, the primary lever arm (depending on the mechanical installation of the system) is its position on the vehicle with respect to the INS body. This lever arm is the fundamental parameter to correct the positions and velocities acquired by the GNSS receiver.

Those parameters are set in the sbg_driver configuration files written in Python



(a) RC INS input parameters



(b) SC19D INS input parameters

Figure 4.14: sbg_driver configuration file. 4.14a for the RC car with all the parameters at 0; 4.14b for the non null parameters coming from the physical measurements done by the Team.

(*.yaml* extension). In figure 4.14 are depicted both the SC19D INS input parameters and the rapid platform parameters. On the radio controlled vehicle, the distances among the localization sensors are in the order of centimetres while the distances between the sensors and the centre of motion of the vehicle are negligible as well. Thus, the design choices were to set all the lever arms and the primary ones to 0 value.

Chapter 5

Results and Discussion

This chapter is dedicated to the results presentation and discussion. The design of the perception pipeline presented has been possible thanks to the experimental data collection and analysis carried out during the development of this thesis work. The results proposed in this chapter come from the experimental validation of the hardware and software components performed on track. Firstly, a first validation of the algorithms proposed is reported using a simulation environment. As analysed in the previous chapters, the modular hardware architecture together with the common software development platform adopted, allowed for a rapid prototyping of the solutions and a testing phase both on the radio controlled and the real car. Although the RC car has been mainly used to rapidly test without the need of setting up the racing prototype, also some tests with the stereo camera positioned at an height of 1.05 m has been done to replicate almost perfectly the real car set up. Unfortunately, during the last months of this thesis, there were not the possibility to put the SC19D on the ground due to the hard work coming from the Squadra Corse PoliTo divisions. In this phase of the autonomous system development, the

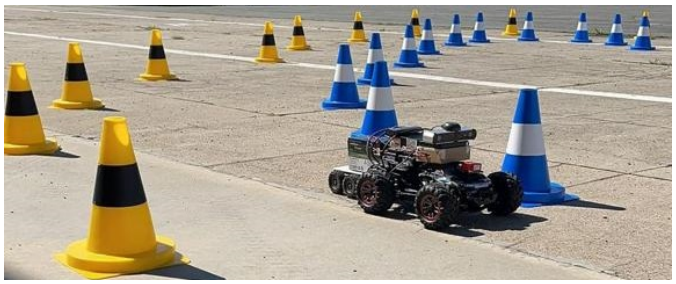


Figure 5.1: Validation Test

car as well as the radio controlled vehicle are driven manually for the collection of the dataset and the validation of the perception pipelines in real-time.

5.1 Validation using Simulation

A computer simulation is a tool devoted to investigating the behaviour of the system taken in consideration. Obviously, the real world must be modelled in such a way it can be constrained in a simulation environment. The main benefits are the faster way to perform the testing phase since they are performed at home allowing to check if small modifications can be beneficial or not. Moreover, working on a software level and not to a hardware one there is no time-consuming preparation, and no ready-to-race vehicles needed. Another benefit is the possibility to change variables and parameters very easily giving the feasibility to decide specifically everything starting from the racetrack and so on. This is the fundamental advantage of the simulation; the simulated scenario is always available making the tests not dependent by the weather conditions and surrounding areas that the real life scenarios does.

The simulation environment in the robotics field has to satisfy the following requirements:

- Real-time simulation
- Sensors modelled close to the real world
- The simulated system is physically truthful

The simulator used to test the perception algorithms is the **EUFS** one. It is a simulation tool implemented in Gazebo by *Edinburgh University Formula Student team*. It allows to select four different track scenarios:

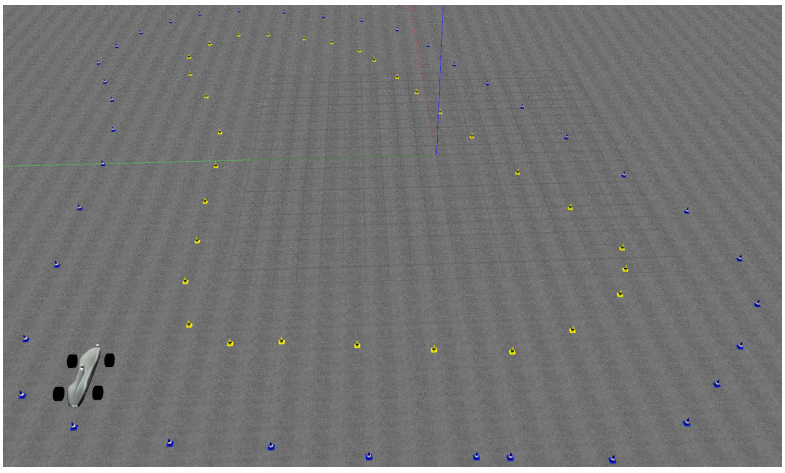


Figure 5.2: The small racetrack

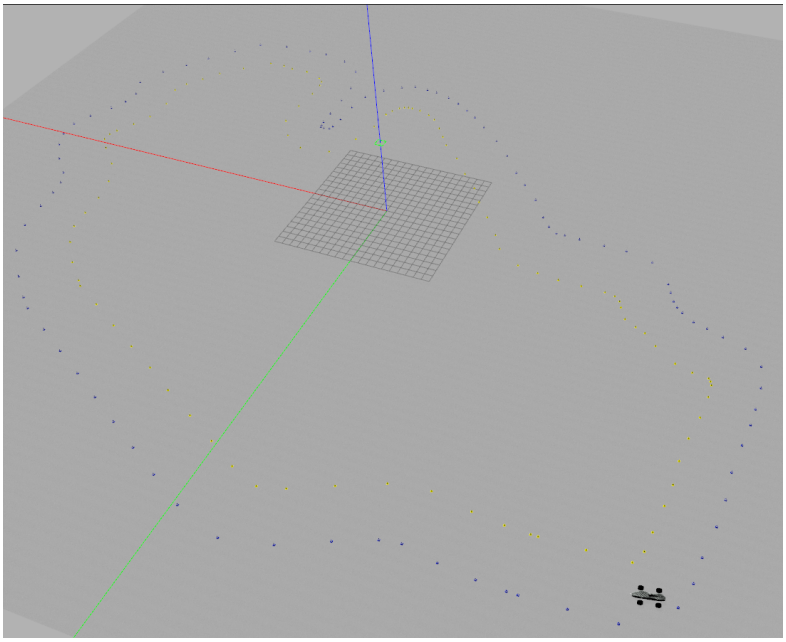


Figure 5.3: The big racetrack

In Figures 5.2 5.3 *small track* and *big track* are shown respectively. They are devoted for making rapid prototyping, while the *skippad* and the *acceleration* tracks are preconfigured tracks which are compliant with the Formula Student competition (Figures 5.4 5.5).

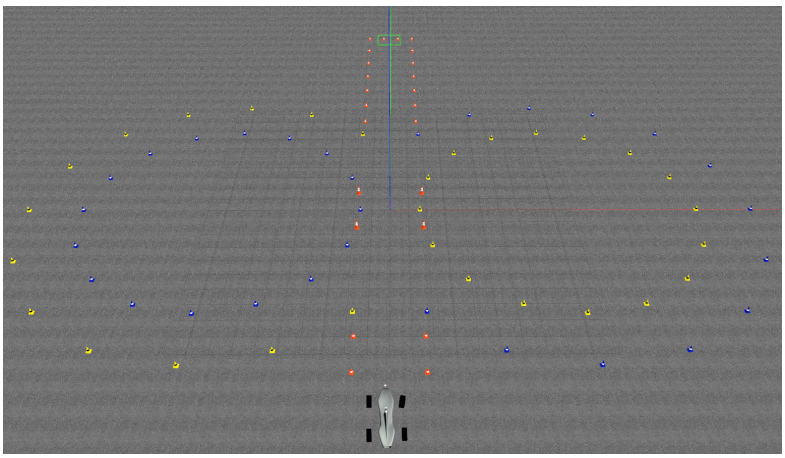


Figure 5.4: The skippad racetrack

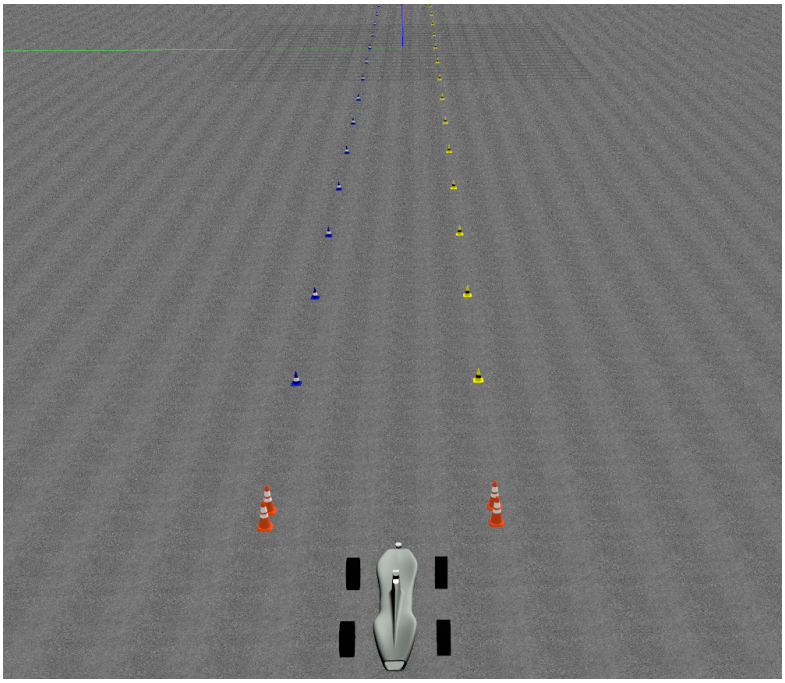


Figure 5.5: The acceleration racetrack

The vehicle model is implemented using the *gazebo_race_car* which allows to carry on the track a custom vehicle model (Figure 5.6). The sensors which are modelled and equipped on the modelled vehicle are:

- Velodyne VLP16
- ZED stereocamera
- IMU
- GNSS

To summarize, the modelled parts on the EUFS simulator are the sensors, the car, and the environment. Indeed, all the autonomous software components are the same for the real test implementation. In our case, the perception pipeline (both with the SSD and the YOLO algorithms) are tested mainly with the acceleration track.

Finally, it is worth pointing out that the input topics used coming from the fictitious ZED stereo camera used in the simulation are modelled almost perfectly. In fact, both of them are modelled without any disturbances and noises that have been carrying on the real world testing phase. In Figure 5.7 the image topics (*zed/left/image_rect_color* and */zed/depth/depth_registered*) are shown

highlighting how far they are from real world in terms of sun exposure or disturbance coming from the reflection phenomenon. In Section 5.2 the real depth image will be show highlighting how much the two depth images coming from the simulation and the real world environment present huge difference. In addition, during this validation test an important limitation has been encountered regarding the depth image view. After some tests has been possible to understand how the depth image works since the cones appear in the image only at distances less than 3 m, this is clearly a decision taken by the simulator owner.

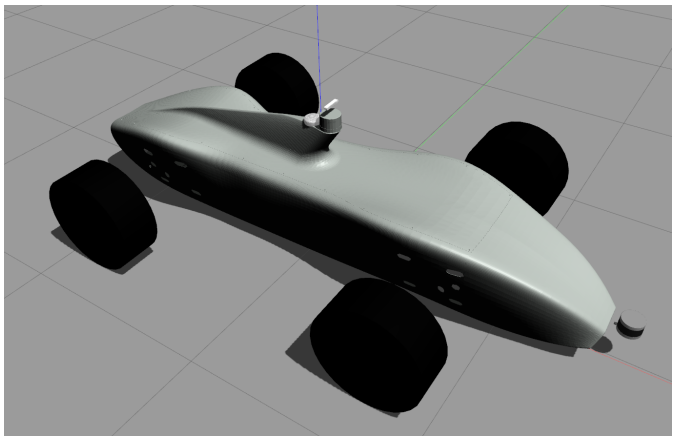
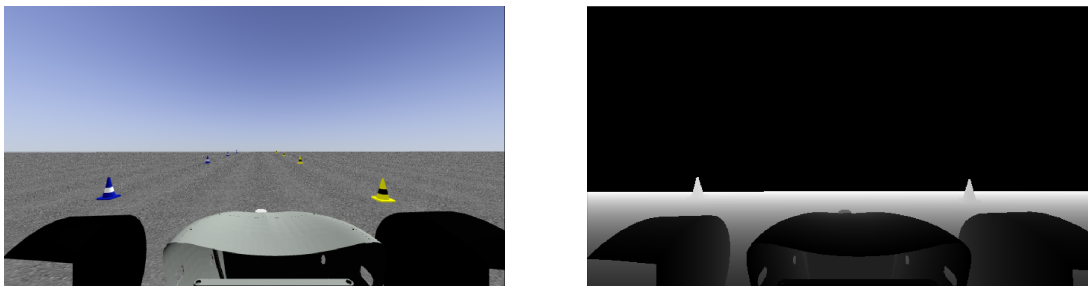


Figure 5.6: The vehicle in the EUFS simulator. It is possible to see the LiDAR in the front wing and the stereo camera in the highest car position



(a) *zed/left/image_rect_color* **(b)** */zed/depth/depth_registered*

Figure 5.7: Example of the input topics in the simulation environment

SSD simulation test

As already analyzed when dealing with the theory about the perception pipeline accomplished by the Single Shot Detector (Chapter 4), all the tasks are performed by a single node, the *perception_pipeline* one. It receives as input topics the images coming from the modelled stereo camera that are: *zed/left/image_rect_color* for the coloured image and *zed/depth/depth_registered* for the depth image. From Figure 4.1, it is worth recalling the two output topics published are: the */perception_cones* topic that is a matrix containing the colour and cones positions needed to accomplish all the perception pipeline, and the */processedImage* topic. This last one has as its sole purpose the visualization on the rviz environment. Therefore, to check directly the algorithm behaviour the rviz tool visualizer is used to publish the */processedImage* topic that it is featured with the bounding boxes around the cones, their respective colours, and the distances between the stereo camera and the cones.

Figure 5.8 represents the */processedImage* topic on rviz. The depth limited distance is taken into account since the vehicle is positioned at a distance smaller than 3 m from a cone.

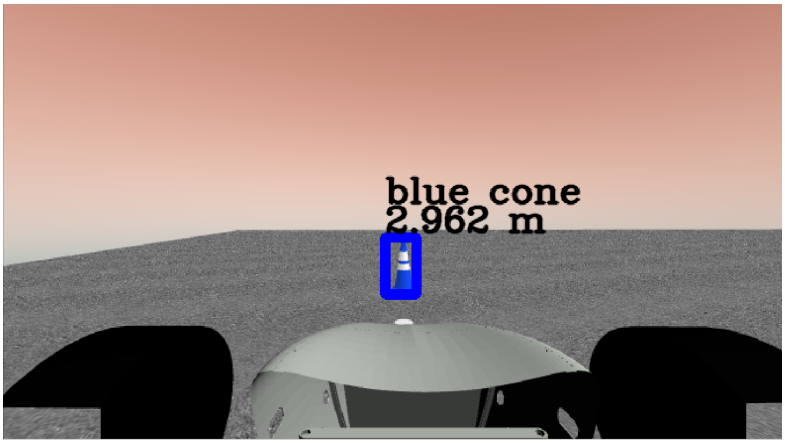


Figure 5.8: */processedImage* visualized on rviz simulation environment

YOLO simulation test

Recalling the yolo block diagram in Figure 4.5, to perform the cone recognition and distance estimation two nodes are needed. The first node is the *darknet_ros* node which publish the */darknet_ros/bounding_boxes* topic, while the *perception_pipeline* node publishes the */perception_cones* and */processedImage* topics as in the SSD case. Both of the two nodes publish topics that can be checked on rviz. Thus, a comparison between these two topics allow to see directly both the cone

detection task and the further merge between it and the depth image. In particular the */darknet_ros/bounding_boxes* topic not only presents the colour detected by the algorithm and the relative bounding box, but also the corresponding probability value. For this reason, the two images are checked at the same time. It could happen that the cone probability is less than 50% (looking at */darknet_ros/bounding_boxes* topic), this means that the depth estimation will not be considered due to the small probability. If this condition happens the */darknet_ros/bounding_boxes* topic shows that the cone is recognized, but in the */processedImage* topic the cone is not highlighted.

Since the depth image detect cones only at a certain distance, this test is performed as well as the SSD simulation test. Figure 5.9 represents on the right image the */darknet_ros/bounding_boxes* topic which detect all the four cone colours correctly. Instead, the */processedImage* topic is on the left image which detect only the closer cones with respect to the vehicle confirming that only the cones under 3 m are recognized.

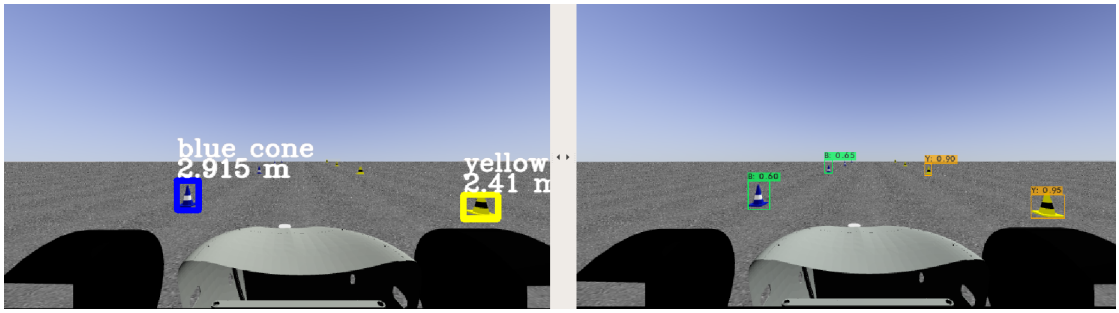


Figure 5.9: On the left the */processedImage* topic. On the right the */darknet_ros/bounding_boxes* topic

Finally, before diving into the real test discussion. It is worth pointing out the small differences among the topics coming from the real ZED stereo camera and the modelled one. Topics are shown in Table 5.1 where it is straightforward to see that the topics come from the real camera have one */zed_node/* more in the path.

	Modelled stereo camera	**ZED stereo camera**
Colour	*zed/left/image_rect_color*	*zed/zed_node/left/image_rect_color*
Depth	*zed/depth/depth_registered*	*zed/zed_node/depth/depth_registered*

Table 5.1: Stereo camera input topics

5.2 Real Test

In this section, the outcomes of the real world have been presented and discussed. Although the SC19D vehicle could not be used for the validation of the algorithm, the testing phase has been carried out at two different stereo camera height positions. The first one is obliged by the physical development of the RC car which set the camera at 0.20 m from the ground. While the second configuration is analyzed with a stereo camera height positioning at 1.05 m. In order to validate the proposed methods, the driving environment is properly structured with traffic cones according to the rules of Formula Student Germany (2019). Each traffic cone has a height equal to 0.325 m and a square base, with a side length equal to 0.228 m. Two main physical issues come from the stereo camera positioning in the radio controlled vehicle, they are:

- Images with shallow depth, since the camera is at the same height of the cones, only the ones in front of the camera have been detected;
- Almost null pitch angle of the camera (Figure 5.10) may lead to have phenomena like direct sun exposure and camera reflections.

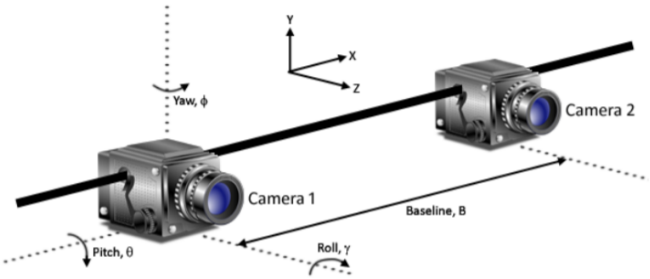


Figure 5.10: angles used for the stereo camera

There are two main possibilities about the validation and the post-processing of the data coming from the algorithms. The first way is to record through *Rosbag* file (explained in Section 3.3.1) the ROS topics coming from the ZED stereo camera (presented in Table 3.1, Subscriber) and saving it in a hard disk directly connected to the embedded computer. It allows to process and test the algorithms by home anytime, saving time since everything is launched in our personal computer. Important to say is the memory used to save the images coming from the camera, after some tests it was depicted that recording the two images for around 2 minutes occupies about 10 Gb of memory resources. Indeed, the second possibility means testing everything in the racetrack and directly see the behaviour of the algorithms. Since the RC car is used to test the whole pipeline (perception, localization and

mapping), to make a post-processing phase could be beneficial to record only the output topics that come from the perception and localization stacks without capturing images that could be too heavy, allowing to test the mapping pipeline at a later stage.

The following discussion is divided in multiple tests done in Aeroclub Torino. In particular, it is divided in two subsections: in subsection 5.2.1 are discussed the cone recognition performed both by SSD and YOLO at the two different heights; subsection 5.2.2 presents the outcomes under different light conditions such as cloudy and sunny days as well as indoor and outdoor. The validation stage has been performed in Aeroclub Torino shown in Figure 5.11.

Figure 5.11: Aeroclub Torino view by Google maps

5.2.1 Scenario 1 - height variation test

The first validation phase is based on the evaluation of the perception pipeline behaviour under two height conditions. Mainly, they are at 20 cm, that is the stereo camera position on the radio controlled vehicle, and 1.05 m that is the camera position on the SC19 car. Thus, two separate phases have been developed based on the vehicle's type used. Moreover, both tests have been analyzing under 6 different distances spanning from 1.5 m to 9 m with a gap of 1.5 m.

Radio controlled height validation Test

This validation phase aims to show the behaviour of the two neural networks on the rapid platform implemented. First, the three cone's types are set in front of the vehicle to check if the cones are recognized properly using the convolutional neural networks both, and secondly, the percentage relative error is calculated between the real cone distance and the measured one. Before analyzing the depth estimation, Figure 5.12 represents the test done setting the cone at 1.5 m, 3 m and 4.5 m. This analysis does not go further the 4.5m distance since the obtained probability for higher distances is always less than the threshold value. The cone recognition performed by the two convolutional neural networks is accurate for our application. About the SSD one, when the cones are detected, 100% of the cones are recognized properly. On the contrary, the Yolo algorithm detects properly the blue and yellow cones, while the orange ones are always not corrected seeing them as yellow cones. The following picture is composed by five different variables, it represents the correctness of the cone recognition. On the x-axis is presented all the cone real distances and which neural network is used. The output is set to 1 when the cone is recognized properly (blue cone sees as blue etc..) while it is 0 when there is a mismatch between them. Then, the graph's points are coloured representing which colour the neural network is detected. For example, looking at the cone recognized by the Yolo at 1.5 m, the orange cone is recognized as a yellow one.

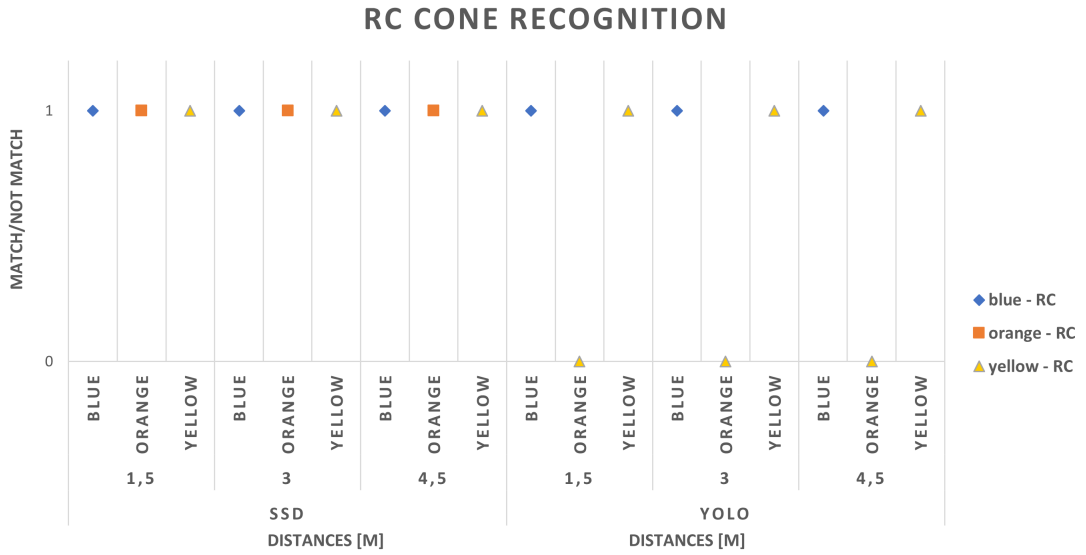


Figure 5.12: Cone recognition with radio-controlled vehicle

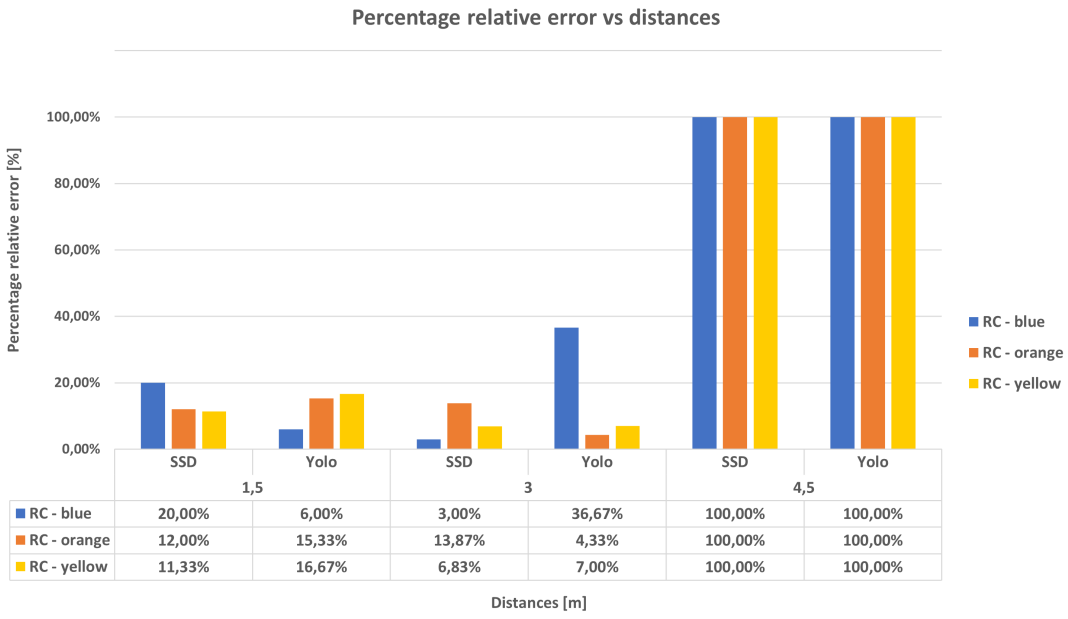
Then, it is tested the matching between the cone recognition and the depth map.

To address it, the absolute and relative errors between the real positioning of the cone and the measured distances are computed. Figure 5.13 represents the single measurements obtained in the Aeroclub with respect to the neural network used.

	1,5m		3m		4,5m	
RC	SSD	Yolo	SSD	Yolo	SSD	Yolo
Yellow	1,67m	1,75m	3,205m	3,21m	0m	0m
Blue	1,8m	1,41m	2,91m	4,1m	0m	0m
Orange	1,68m	1,73m	3,416m	2,87m	0m	0m

Figure 5.13: Distance measurements at different distances

In Figure 5.14 the histogram has been developed to present the percentage relative error in function of the distances. As already explained, the physical problem of the stereo camera position gives the opportunity to detect the cones only at the first distances. In fact, at 4.5 m the cones are recognized with a probability less than the threshold value which do not allow to measure the cone distances. For this reason, the percentage relative error is equal to 100% at measurements higher than 4 m. On the contrary, the measurements obtained at the first two distances are accurate with a small error error that will be easily neglected afterward.



	SSD	Yolo	SSD	Yolo	SSD	Yolo
	1,5		3		4,5	
RC - blue	20,00%	6,00%	3,00%	36,67%	100,00%	100,00%
RC - orange	12,00%	15,33%	13,87%	4,33%	100,00%	100,00%
RC - yellow	11,33%	16,67%	6,83%	7,00%	100,00%	100,00%

Figure 5.14: Histogram about Radio controlled vehicle

Squadra Corse height validation Test

In this scenario the ZED stereo camera is set at 1.05 m from the ground. The camera positioning allow a deeper field of view with respect to the RC car, for this purpose, the cones are positioned till 9 m from the stereo camera. Moreover, the stereo camera position and its orientation allow to perform a better cone recognition since the background of the cones is the monochrome road.

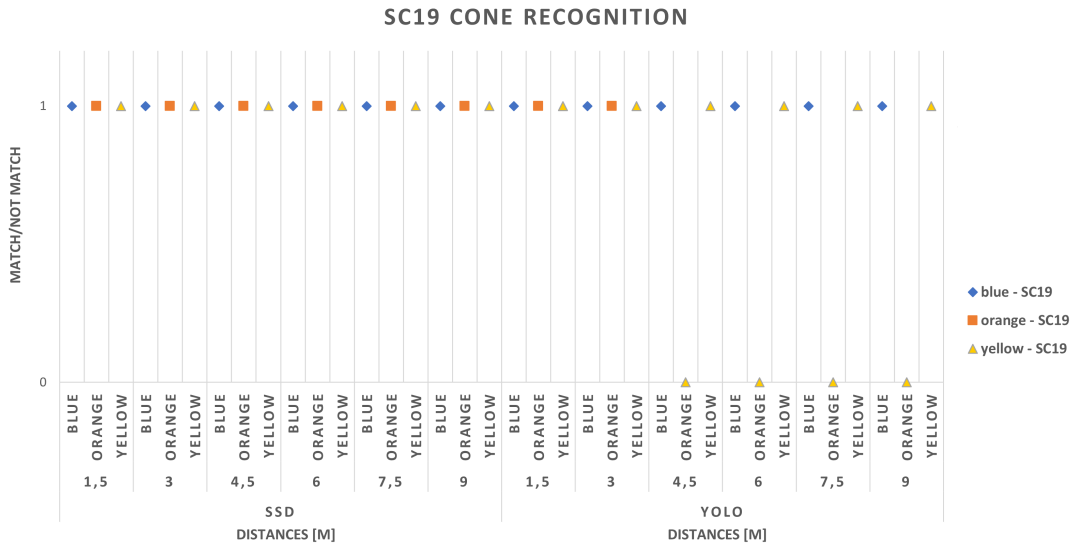


Figure 5.15: Cone recognition with Squadra Corse racecar

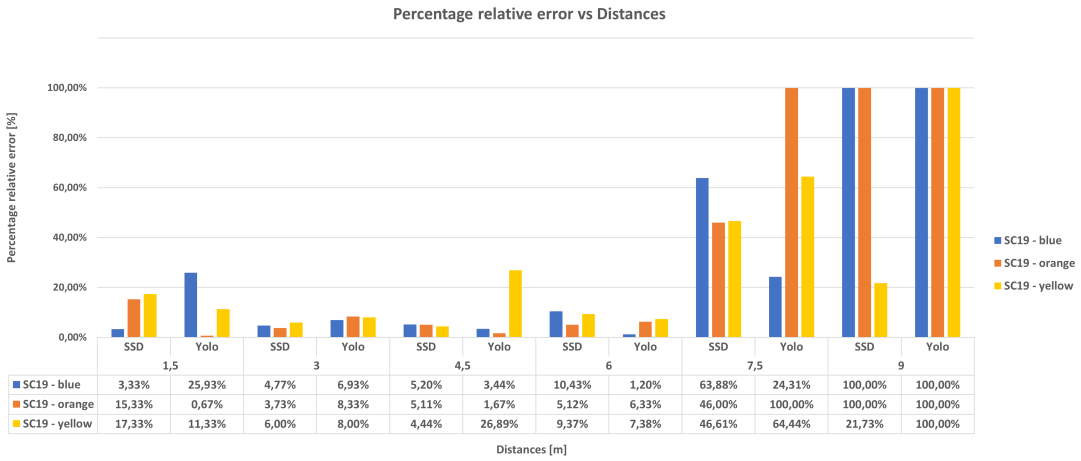
As in the previous case, the SSD algorithm performs a perfect cone recognition while the Yolo algorithm results are improved from the radio controlled vehicle test. Indeed, at least in the first two tests using Yolo the orange cones are properly recognized while in the other ones the problem still remain. Analyzing the pure cone recognition task as in Figure 5.15, both of the CNNs perform properly for the cone detection task. Figure 5.16 shows the measurements obtained by matching the bounding boxes and the depth image. The histogram in Figure 5.17 represents

	1,5m		3m		4,5m		6m		7.5m		9m	
SC19	SSD	Yolo	SSD	Yolo	SSD	Yolo	SSD	Yolo	SSD	Yolo	SSD	Yolo
Yellow	1,76m	1,67m	3,18m	3,24m	4.7m	5.71m	5.438m	6.443m	10.996m	12.333m	10.956m	0m
Blue	1,45m	1,889m	3.143m	3.208m	4.266m	4.345m	6.626m	5.928m	12.291m	9.323m	0m	0m
Orange	1,73m	1,51m	3,112m	3.25m	4.27m	4.575m	6.307m	6.38m	10.95m	0m	0m	0m

Figure 5.16: Distance measurements at different distances

the percentage relative error in function of the real cone distances from the stereo camera. The 1.5 m measurement is tested just for completeness; however, since the

stereo camera is placed on the main roll hoop (above the driver's seat) as shown in Figure 3.11, analyzing 1.5 m distance is like having the cone attached to the vehicle. From the histogram it is possible to see that the optimal outcomes are obtained from 1.5 m to 7.5 m. Also in this case, the small errors can be neglected using some techniques that will be presented in Chapter 6. In this testing stage,



	SSD 1,5	Yolo 1,5	SSD 3	Yolo 3	SSD 4,5	Yolo 4,5	SSD 6	Yolo 6	SSD 7,5	Yolo 7,5	SSD 9	Yolo 9
SC19 - blue	3,33%	25,93%	4,77%	6,93%	5,20%	3,44%	10,43%	1,20%	63,88%	24,31%	100,00%	100,00%
SC19 - orange	15,33%	0,67%	3,73%	8,33%	5,11%	1,67%	5,12%	6,33%	46,00%	100,00%	100,00%	100,00%
SC19 - yellow	17,33%	11,33%	6,00%	8,00%	4,44%	26,89%	9,37%	7,38%	46,61%	64,44%	21,73%	100,00%

Figure 5.17: Histogram about Squadra Corse race car

a final consideration can be done about the frame-rate of the two algorithms. In fact, during the tests the frequency was computed locally in the script calculating the time that pass from acquiring the stereo camera images till the end of the

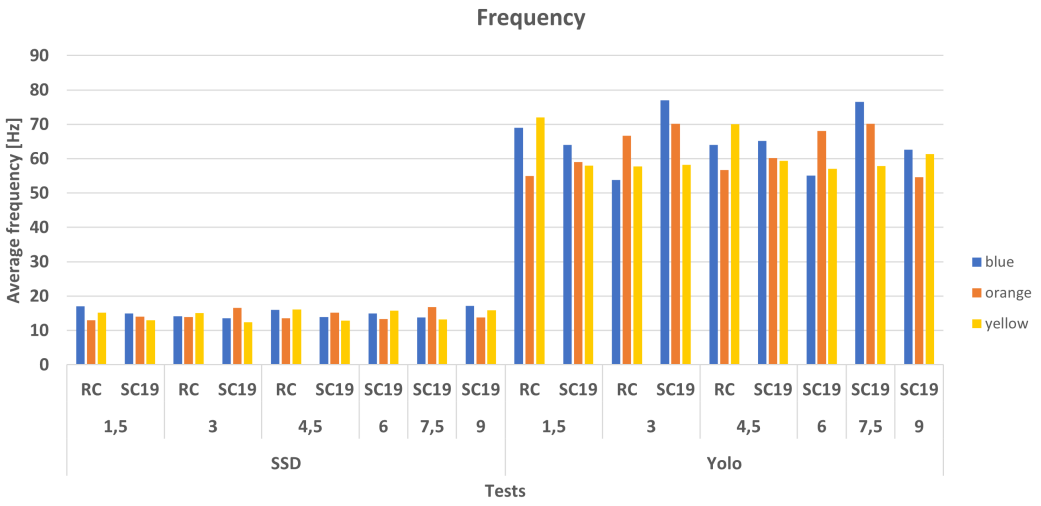


Figure 5.18: Average frequency obtained for each test

computation. Looking in this direction, the improvement obtained by the Yolo algorithm is almost quadruple the SSD frequency. The frame-rate analysis is based on saving ten consecutive frequency measurements and computing the mean value for each test. Thus, Figure 5.18 represents the average frequency [Hz] obtained for each test.

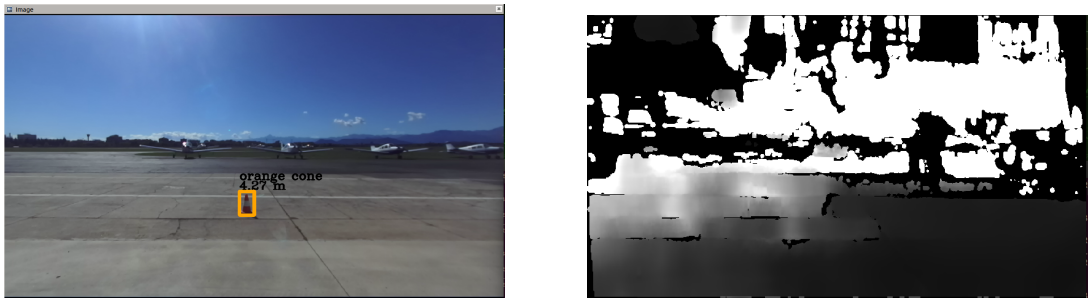
To have a direct comparison between the frame-rate of the two convolutional neural networks the average frequencies obtained in the previous step have been averaged again between the CNNs considered. Table 5.2 presents the final frequency average of the two convolutional neural networks obtained during the validation phase.

Neural networks	**Frequency**
Single Shot Detector	14.62 Hz
You Only Look Once	62.93 Hz

Table 5.2: Frequency obtained using the two CNNs

5.2.2 Scenario 2 - environmental condition

This validation phase is based on the evaluation of the perception pipeline behaviour in different environment condition. As already analyzed in the previous section, from distances larger than 7.5 m the measured distances become inaccurate and not stable. The reason might be depicted by the depth image acquired by the ZED stereo camera. Therefore, Figure 5.19 shows the depth image obtained during a standard validation phase performed in a sunny day at the Aeroclub together with the /processedImage obtained using the SSD algorithm. In this outdoor condition, the camera is subject directly to the sun exposure giving a noise depth image. This issue leads to an error increases with increasing distance. Recalling



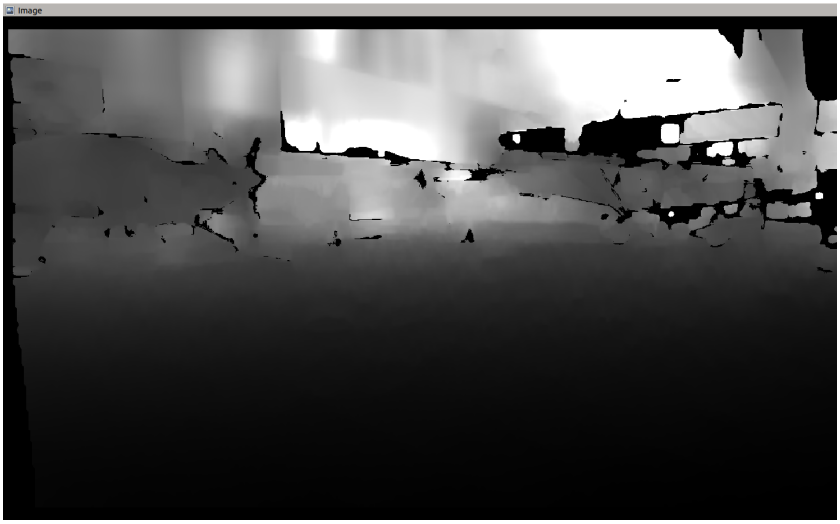
(a) /processedImage **(b)** Depth image

Figure 5.19: SSD outcome and related depth image. Outdoor test

the stereocamera based perception algorithm explained in Section 4.1, the match between the bounding boxes and the depth image has been addressed by calculating the center point of the bounding box and looking at the corresponding point in the depth image. For this reason, since only a point is considered to estimate the distance, the estimated distance is prone to be unstable in the case of a noisy depth image.

To support this thesis, some tests inside an hangar of the Aeroclub has been performed just to see the depth map behaviour. Recalling the outcomes come from the first scenario, the distance is computed accurately till 6 m, while going up to this distance it becomes unstable and finally not acquired. Thus, two cones were

(a) Left */processedImage*. Right */darknet_ros/bounding_boxes*



(b) Depth image

Figure 5.20: Yolo outcome and related depth image. Indoor test

positioned at a distance of 10m in the already mentioned indoor place. The result shows a stable measurements of the cones that can be justified by the depth image obtained in this case. Comparing Figure 5.19b with Figure 5.20b it is immediate to see how the indoor depth image has a very smooth form with respect to the one obtained outside. To fix this unstable depth image problem, the team has been starting to work to implement a redundant perception task through the use of LiDAR sensor. The idea is to interpolate the data coming from the two sensors giving more importance to the LiDAR measurement that should be more accurate with respect to the stereo camera one.

Finally, the last test was performed during a cloudy day. Although in this particular condition the algorithms do not have different performance from the one obtained during sunny days, the depth image is less noise allowing a more stable distance computation at distances higher than 6 m.

Figure 5.21: Test during cloudy day

Chapter 6

Conclusion and Future Works

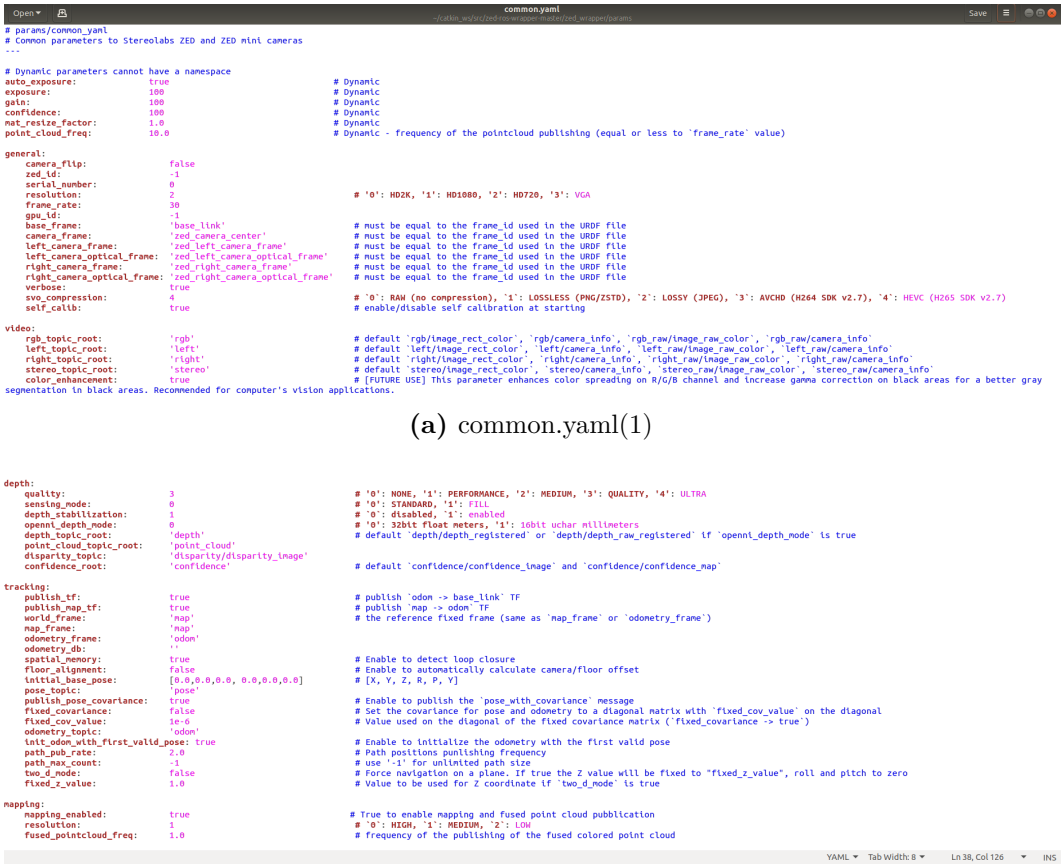
Nowadays, research interest towards the autonomous driving world has becoming more and more popular aiming to address hard challenges as safety and good performances of the future automated vehicle. Vehicle's perception represents one of the most relevant and challenging issues for autonomous vehicles. In this experimental project presented, the Perception pipeline for an autonomous race car has been deeply analyzed. This project has been divided in multiple stages. Firstly, regarding the software architecture design, the Machine Learning world has been deeply investigated looking for the best Convolutional Neural Networks in the object detection panorama. For this purpose, this approach proposed and compared the application of a Single Shot Detector (SSD) and a You Only Look Once (YOLO) neural networks. The second design stage has been devoted to the hardware architecture design, analyzing the optimal sensor's configuration and creating a rapid platform to perform validation phase without using the Squadra Corse PoliTo race car. The solution presented shows a reliable cone recognition with the two neural networks both. However, the main differences between the two configurations can be depicted checking the time that they need to complete the whole algorithm. While the SSD frequency reaches at most 15 Hz, the YOLO one is able to have the mean value frequency more than 60 Hz. Thus, according to the theoretical comparison done in Chapter 2, in our application the Yolo algorithm is the one that better accomplish the cone recognition and its distance. Then, due to the structure of the depth computation, the cone distance measurements have the same behaviour in both the neural networks. Although the rapid platform has been useful to rapidly test the proposed algorithms, some limitations due to its physical dimensions have been carried out. In particular, the narrow depth allow the cone recognition just for the first meters from the radio controller vehicle, whereas the

camera orientation can aim at the sun leading to obtain a not robust depth image due to sun exposure and reflection phenomenon.

In the future development of this work, the most important task is to re-train the YOLO algorithm with a huge dataset of images. In this way, the issue about the orange cone recognition will be fixed obtaining an accurate neural network. Regarding the depth computation, our perception division has been working on a redundant technique exploiting by the combination of the already used stereo camera with the LiDAR sensor. As explained in Chapter 4, after cone recognition and the creation of the corresponding bounding box, the bounding box center point is calculated mathematically and it is compared with the distance value obtained by the depth image at this precise image position. This measurement can be easily affected by some drawbacks that are proper of the stereo camera such as accomplish tasks under varying light and visibility conditions and in scenes with a high dynamic range. Therefore, the LiDAR represents a recent technology that accurately computes distance to objects by measuring the flight-time of multiple laser pulses. However, the LiDAR has some drawbacks, its limited vertical resolution is easily solved by the LiDAR's position since it is positioned on the front wing of the race car. In addition, it is not suitable for detecting small objects placed at great distances and the measurements could be strongly affected by the weather condition. However, using a redundant configuration with multiple sensors lead to a better distance estimation. Finally, the two cone measurements coming from the LiDAR and the stereo camera will be interpolated giving more importance to the LiDAR measurement, that is the most accurate one.

Appendix A

Camera Calibration



(a) common.yaml(1)

(b) common.yaml(2)

Figure A.1: common.yaml parameter configuration

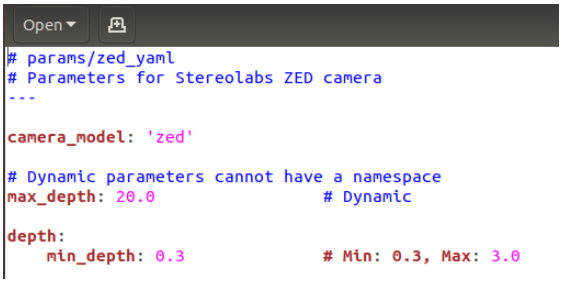


Figure A.2: zed.yaml

Appendix B

SSD Code

perception_pipeline.py

```
#!/usr/bin/env python
import cv2
import numpy as np
import tensorflow as tflow
from matplotlib import pyplot as plt
from numpy import array
from utils import visualization_utils as vis_util
from utils import label_map_util
from ConeDetection import *
from cone_img_processing2 import *
import os
import rospy
from std_msgs.msg import String
from sensor_msgs.msg import Image , PointCloud2
from cv_bridge import CvBridge, CvBridgeError
import time
import copy
from visualization_msgs.msg import Marker
from geometry_msgs.msg import Quaternion, Pose, Point, Vector3
from nav_msgs.msg import Odometry
from geometry_msgs.msg import PoseWithCovariance, Pose
from geometry_msgs.msg import PointStamped
import tf
from rospy.numpy_msg import numpy_msg
from rospy_tutorials.msg import Floats

from tensorflow.compat.v1 import ConfigProto
from tensorflow.compat.v1 import InteractiveSession

config = ConfigProto()
config.gpu_options.allow_growth = True
session = InteractiveSession(config=config)

cy = 352.49
fy = 672.55
cx = 635.18
fx = 672.55
threshold_cone = 0.5

PATH_TO_CKPT = os.path.dirname(os.path.realpath(__file__)) + '/frozen_cone_graph.pb'
PATH_TO_LABELS = os.path.dirname(os.path.realpath(__file__)) + '/label_map.pbtxt'

NUM_CLASSES = 1

detection_graph = tflow.Graph()
with detection_graph.as_default():
  od_graph_def = tflow.GraphDef()
  with tflow.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
    serialized_graph = fid.read()
    od_graph_def.ParseFromString(serialized_graph)
    tflow.import_graph_def(od_graph_def, name='')

label_map = label_map_util.load_labelmap(PATH_TO_LABELS)
categories = label_map_util.convert_label_map_to_categories(label_map, max_num_classes=NUM_CLASSES, use_display_name=True)
category_index = label_map_util.create_category_index(categories)

class Frame:
    def __init__(self):
        self.image = []
        self.flagImage = False
        self.depth = []
        self.flagDepth = False

lastFrame = Frame()
```

```
class Cone:
    def __init__(self):
        self.color = 0
        self.x = 0
        self.y = 0

def publish_cones(local_map, transform_to_map=False):
    if local_map.shape[0] == 0:
        return

    cone_publisher = rospy.Publisher('/perception_cones', numpy_msg(Floats), queue_size=10)
    transformed_cones = np.array([])

    for cone in local_map:
        conePoint_local = PointStamped()
        conePoint_local.header.frame_id = 'zed_left_camera'
        conePoint_local.point.x = cone[2]
        conePoint_local.point.y = -cone[0]
        conePoint_local.point.z = -cone[1]
        local_cone = np.array([conePoint_local.point.x, conePoint_local.point.y, conePoint_local.point.z, float(cone[3])], dtype=np.float32).reshape(1,4)
        if transform_to_map:
            print()
        else:
            if transformed_cones.shape[0] > 0:
                transformed_cones = np.vstack((transformed_cones, local_cone))
            else:
                transformed_cones = local_cone
    cone_publisher.publish(transformed_cones.flatten())

def callback_show(image_message):
    tick = time.time()
    bridge = CvBridge()
    image = bridge.imgmsg_to_cv2(image_message, desired_encoding="passthrough")
    cv2.imshow('object detection', image)
    cv2.waitKey(1)
    tock = time.time() - tick

def callback_storage_image(image_message):
    bridge = CvBridge()
    global lastFrame
    lastFrame.image = bridge.imgmsg_to_cv2(image_message, desired_encoding="passthrough")
    lastFrame.flagImage = True

def callback_depth(image_message):
    bridge = CvBridge()
    global lastFrame
    lastFrame.depth = bridge.imgmsg_to_cv2(image_message, desired_encoding="passthrough")
    lastFrame.flagDepth = True
```

```
def mainLoop():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("/zed/zed_node/left/image_rect_color", Image, callback_storage_image)
    rospy.Subscriber("/zed/zed_node/depth/depth_registered", Image, callback_depth)
    processedImage = rospy.Publisher("/processedImage", Image, queue_size=10)

    global listener
    listener = tf.TransformListener()

    r = rospy.Rate(100) # Hz
    with detection_graph.as_default():
        with tflow.Session(graph=detection_graph) as sess:
            while not rospy.is_shutdown():
                tick = time.time()
                global lastFrame
                processFrame = copy.deepcopy(lastFrame)

                if processFrame.flagImage == True and processFrame.flagDepth == True:
                    image_np = processFrame.image
                    image_np = image_np[:,:,:3]
                    image_np_expanded = np.expand_dims(image_np, axis=0)
                    image_tensor = detection_graph.get_tensor_by_name('image_tensor:0')
                    boxes = detection_graph.get_tensor_by_name('detection_boxes:0')
                    scores = detection_graph.get_tensor_by_name('detection_scores:0')
                    classes = detection_graph.get_tensor_by_name('detection_classes:0')
                    num_detections = detection_graph.get_tensor_by_name('num_detections:0')
                    (boxes, scores, classes, num_detections) = sess.run(
                        [boxes, scores, classes, num_detections],
                        feed_dict={image_tensor: image_np_expanded})
                    boxes = np.squeeze(boxes)
                    scores = np.squeeze(scores)

                    width = image_np.shape[1]
                    height = image_np.shape[0]
                    output_img = image_np.copy()
                    center_x = []
                    center_y = []
                    center_z = []
                    center_color =[]

                    for i in range(boxes.shape[0]):

                        if np.all(boxes[i] == 0) or scores[i] < threshold_cone:
                            continue
                        b = boxes[i]
                        box_width = np.abs(float(b[3])-float(b[1]))
                        box_height  = np.abs(float(b[2])-float(b[0]))
                        x = int(b[1] * width)
                        y = int(b[0] * height)
                        h = int(box_height * height)
                        w = int(box_width * width)

                        candidate = image_np[y:y+h, x:x+w]
                        y = y + 1
                        result = detectCone1(candidate)

                        square_size = 5
                        depth_square = lastFrame.depth[int(-square_size+round(y+h/2)):int(square_size+round(y+h/2)),int(-square_size+round(x+w/2)):int(square_size+round(x+w/2))]
                        depth_square = copy.copy(depth_square)
                        bad_I = np.argwhere(np.isnan(depth_square))
                        depth_square[bad_I[:,0],bad_I[:,1]] = 0
                        bad_I = np.argwhere(np.isinf(depth_square))
                        depth_square[bad_I[:,0],bad_I[:,1]] = 0

                        valid_index = np.nonzero(depth_square)
```

```
                    if valid_index[0].shape[0] == 0 or valid_index[1].shape[0] == 0:
                        continue

                    z = np.mean(depth_square[valid_index[0], valid_index[1]])

                    try:
                        y_w = (round(y+h/2) - cy) / fy * z
                        x_w = (round(x+w/2) - cx) / fx * z
                    except:
                        continue

                    center_x.append(x_w)
                    center_y.append(y_w)
                    center_z.append(z)
                    center_color.append(result)

                    if result == 0:
                        print("Yellow Cone")
                        cv2.rectangle(output_img, (int(b[1] * width),int(b[0] * height)), (x+w,y+h), (0, 255, 255), 7)
                        cv2.putText(output_img, 'yellow cone', (int(b[1] * width),int(b[0] * height)-30), cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 0), 2, cv2.LINE_AA)
                        cv2.putText(output_img,  str(round(z,3))+" m", (int(b[1] * width),int(b[0] * height)-5), cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 0), 2, cv2.LINE_AA)

                    if result == 1:
                        print("Blue Cone")
                        cv2.rectangle(output_img, (int(b[1] * width),int(b[0] * height)), (x+w,y+h), (255, 0, 0), 7)
                        cv2.putText(output_img, 'blue cone', (int(b[1] * width),int(b[0] * height)-30), cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 0), 2, cv2.LINE_AA)
                        cv2.putText(output_img,  str(round(z,3))+" m", (int(b[1] * width),int(b[0] * height)-5), cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 0), 2, cv2.LINE_AA)
                    if result == 2:
                        print("Orange Cone")
                        cv2.rectangle(output_img, (int(b[1] * width),int(b[0] * height)), (x+w,y+h), (0,165,255), 7)
                        cv2.putText(output_img, 'orange cone', (int(b[1] * width),int(b[0] * height)-30), cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 0), 2, cv2.LINE_AA)
                        cv2.putText(output_img,  str(round(z,3))+" m", (int(b[1] * width),int(b[0] * height)-5), cv2.FONT_HERSHEY_COMPLEX, 1, (0, 0, 0), 2, cv2.LINE_AA)

                processedImage.publish(CvBridge().cv2_to_imgmsg(output_img))
                r.sleep()
                local_map = np.array((center_x,center_y,center_z,center_color)).T
                publish_cones(local_map)
                tock = time.time() - tick
                print("FPS: "+str(1/tock))
                processFrame.flagImage = False
                processFrame.flagPointCloud = False

if __name__ == '__main__':
    mainLoop()
```

Appendix C

YOLOv4 Code

yolo_subscriber.py

```
#!/usr/bin/env python
import cv2
import numpy as np
import os
import datetime
import copy

#tensorflow imports
import tensorflow as tflow
from utils import visualization_utils as vis_util
from utils import label_map_util

#ros imports
import rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
#import for the yolo
from darknet_ros_msgs.msg import BoundingBoxes
#import for the rviz visualization markers
from visualization_msgs.msg import Marker
from geometry_msgs.msg import Pose
#Pointstamp is a point with reference coordinates and a timestamp for each point
from geometry_msgs.msg import PointStamped
from rospy.numpy_msg import numpy_msg
from rospy_tutorials.msg import Floats

import time

##########needed only for nvidia RTX #############
from tensorflow.compat.v1 import ConfigProto
from tensorflow.compat.v1 import InteractiveSession

config = ConfigProto()
config.gpu_options.allow_growth = True
session = InteractiveSession(config=config)

#Intrinsic camera parameters
cy = 352.49
fy = 672.55
cx = 635.18
fx = 672.55

start = 0

threshold_cone = 0.5

class Frame:
    def __init__(self):
        self.image = []
        self.flagImage = False
        self.depth = []
        self.flagDepth = False
        self.Class = []
        self.xmin = []
        self.xmax = []
        self.ymin = []
        self.ymax = []
        self.probability = []
        self.flagProcessed = False
        self.flagReady = False

lastFrame = Frame()
cone_publisher = rospy.Publisher('/perception_cones', numpy_msg(Floats), queue_size=30)
publishToYolo = rospy.Publisher("/sync_img", Image,queue_size=10)
```

```
def publish_cones(local_map, transform_to_map=False):
    global cone_publisher
    if local_map.shape[0] == 0:
        return

    transformed_cones = np.array([])

    for cone in local_map:
        conePoint_local = PointStamped()
        conePoint_local.header.frame_id = 'zed_left_camera'
        conePoint_local.point.x = cone[0]
        conePoint_local.point.y = cone[1]
        conePoint_local.point.z = cone[2]

        local_cone = np.array([conePoint_local.point.x, conePoint_local.point.y, conePoint_local.point.z, float(cone[3])], dtype=np.float32).reshape(1,4)

        if transform_to_map:
            print()
        else:
            if transformed_cones.shape[0] > 0:
                transformed_cones = np.vstack((transformed_cones, local_cone))
            else:
                transformed_cones = local_cone

    cone_publisher.publish(transformed_cones.flatten())

#call back function to store image in class object
def callback_storage_image(image_message):
    global publishToYolo
    bridge = CvBridge()
    global lastFrame
    global start
    start = time.time()

    if lastFrame.flagImage == False:
        lastFrame.image = bridge.imgmsg_to_cv2(image_message, desired_encoding="passthrough")
        lastFrame.flagImage = True

    publishToYolo.publish(image_message)

#call back function to store depth in class object
def callback_depth(image_message):
    bridge = CvBridge()
    global lastFrame
    if lastFrame.flagDepth == False:
        lastFrame.depth = bridge.imgmsg_to_cv2(image_message, desired_encoding="passthrough")
        lastFrame.flagDepth = True
```

```
        def callback_yolo(data):
            global lastFrame
            if lastFrame.flagProcessed == False:
                lastFrame.flagProcessed = True
                lastFrame.Class = []
                lastFrame.xmin = []
                lastFrame.xmax = []
                lastFrame.ymin = []
                lastFrame.ymax = []
                lastFrame.probability = []
                for i in range(len(data.bounding_boxes)):
                    lastFrame.Class.append(data.bounding_boxes[i].Class)
                    lastFrame.xmin.append(data.bounding_boxes[i].xmin)
                    lastFrame.xmax.append(data.bounding_boxes[i].xmax)
                    lastFrame.ymin.append(data.bounding_boxes[i].ymin)
                    lastFrame.ymax.append(data.bounding_boxes[i].ymax)
                    lastFrame.probability.append(data.bounding_boxes[i].probability)

                selection = [len(lastFrame.xmin), len(lastFrame.xmax), len(lastFrame.ymin), len(lastFrame.ymax)]
                shortest_list = min(selection)
                lastFrame.xmin = lastFrame.xmin[0:shortest_list]
                lastFrame.xmax = lastFrame.xmax[0:shortest_list]
                lastFrame.ymin = lastFrame.ymin[0:shortest_list]
                lastFrame.ymax = lastFrame.ymax[0:shortest_list]
                lastFrame.probability = lastFrame.probability[0:shortest_list]
                lastFrame.Class = lastFrame.Class[0:shortest_list]
                lastFrame.flagReady = True


        def listener()
            rospy.init_node('listener', anonymous=True)
            rospy.Subscriber("/zed/zed_node/left/image_rect_color", Image, callback_storage_image)
            rospy.Subscriber("/zed/zed_node/depth/depth_registered", Image, callback_depth)
            rospy.Subscriber("/darknet_ros/bounding_boxes", BoundingBoxes, callback_yolo)
            processedImage = rospy.Publisher("/processedImage", Image, queue_size=10)
            r = rospy.Rate(100) # Hz
            global start
            while not rospy.is_shutdown():
                global lastFrame
                if lastFrame.flagImage == True and lastFrame.flagDepth == True and lastFrame.flagReady == True:
                    processFrame = copy.deepcopy(lastFrame)
                    if len(processFrame.probability) == 0 :
                        continue
                    image_np = processFrame.image
                    output_img = image_np.copy()
                    width = image_np.shape[1]
                    height = image_np.shape[0]
                    center_x = []
                    center_y = []
                    center_z = []
                    center_color = []
```

```
            for i in range(len(processFrame.probability)):
                if processFrame.probability[i] < threshold_cone:
                    continue

                box_width = np.abs(float(processFrame.xmax[i])-float(processFrame.xmin[i]))
                box_height  = np.abs(float(processFrame.ymax[i])-float(processFrame.ymin[i]))

                x = int(processFrame.xmin[i]) #* width)
                y = int(processFrame.ymin[i]) #* height)
                h = int(box_height) #* height)
                w = int(box_width) #* width)

                #print("xy_pixel",x,y,h,w)
                #print(lastFrame.depth)
                # Taking 10 pixel square
                square_size = 5
                try:
                    depth_square = processFrame.depth[int(-square_size+round(y+h/2)):int(square_size+round(y+h/2)),int(-square_size+round(x+w/2)):int(square_size+round(x+w/2))]
                    bad_I = np.argwhere(np.isnan(depth_square))
                    depth_square[bad_I[:,0],bad_I[:,1]] = 0
                    bad_I = np.argwhere(np.isinf(depth_square))
                    depth_square[bad_I[:,0],bad_I[:,1]] = 0

                    valid_index = np.nonzero(depth_square)
                    if valid_index[0].shape[0] == 0 or valid_index[1].shape[0] == 0:
                        continue

                    z = np.mean(depth_square[valid_index[0], valid_index[1]])
                except:
                    print("depth not valid")

                try:
                    y_w = (round(y+h/2) - cy) / fy * z
                    x_w = (round(x+w/2) - cx) / fx * z
                except:
                    continue









                result = 1
                if processFrame.Class[i] == "R":
                    result = 0
                elif processFrame.Class[i] == "B":
                    result = 1
                elif processFrame.Class[i] == "Y":
                    result = 2

                center_x.append(x_w)
                center_y.append(y_w)
                center_z.append(z)
                center_color.append(result)

                if result == 2:
                    print("Yellow Cone")
                    cv2.rectangle(output_img, (int(processFrame.xmin[i]), int(processFrame.ymin[i])),(int(processFrame.xmax[i]), int(processFrame.ymax[i])), (0, 255, 255), 7)
                    cv2.putText(output_img, 'yellow cone', (int(processFrame.xmin[i]), int(processFrame.ymin[i])-30), cv2.FONT_HERSHEY_COMPLEX, 1, (255, 255, 255), 2, cv2.LINE_AA)
                    cv2.putText(output_img, str(round(z,3))+" m", (int(processFrame.xmin[i]), int(processFrame.ymin[i])-5), cv2.FONT_HERSHEY_COMPLEX, 1, (255, 255, 255), 2, cv2.LINE_AA)
                if result == 1:
                    print("Blue Cone")
                    cv2.rectangle(output_img, (int(processFrame.xmin[i]), int(processFrame.ymin[i])),(int(processFrame.xmax[i]), int(processFrame.ymax[i])), (255, 0, 0), 7)
                    cv2.putText(output_img, 'blue cone', (int(processFrame.xmin[i]), int(processFrame.ymin[i])-30), cv2.FONT_HERSHEY_COMPLEX, 1, (255, 255, 255), 2, cv2.LINE_AA)
                    cv2.putText(output_img, str(round(z,3))+" m", (int(processFrame.xmin[i]), int(processFrame.ymin[i])-5), cv2.FONT_HERSHEY_COMPLEX, 1, (255, 255, 255), 2, cv2.LINE_AA)
                if result == 0:
                    print("Orange Cone")
                    cv2.rectangle(output_img, (int(processFrame.xmin[i]), int(processFrame.ymin[i])),(int(processFrame.xmax[i]), int(processFrame.ymax[i])), (0, 165, 255), 7)
                    cv2.putText(output_img, 'orange cone', (int(processFrame.xmin[i]), int(processFrame.ymin[i])-30), cv2.FONT_HERSHEY_COMPLEX, 1, (255, 255, 255), 2, cv2.LINE_AA)
                    cv2.putText(output_img, str(round(z,3))+" m", (int(processFrame.xmin[i]), int(processFrame.ymin[i])-5), cv2.FONT_HERSHEY_COMPLEX, 1, (255, 255, 255), 2, cv2.LINE_AA)

            processedImage.publish(CvBridge().cv2_to_imgmsg(output_img))
            r.sleep()
            local_map = np.array((center_x,center_y,center_z,center_color)).T
            publish_cones(local_map)
            lastFrame = Frame()
            stop = time.time()
            frame_time = stop - start
            print("frame rate:", 1/frame_time)


if __name__ == '__main__':
    listener()
```

Bibliography

[1] James M. Anderson, Nidhi Kalra, Karlyn D. Stanley, Paul Sorensen, Constantine Samaras, and Oluwatobi A. Oluwatola. «Brief History and Current State of Autonomous Vehicles». In: *Autonomous Vehicle Technology: A Guide for Policymakers*. RAND Corporation, 2014, pp. 55–74. ISBN: 9780833083982. URL: http://www.jstor.org/stable/10.7249/j.ctt5hhwgz.11 (cit. on p. 1).

[2] Swaroop Darbha, S. Konduri, and Prabhakar Pagilla. «Benefits of V2V Communication for Autonomous and Connected Vehicles». In: *IEEE Transactions on Intelligent Transportation Systems* 20 (Mar. 2018). DOI: 10.1109/TITS.2018.2859765 (cit. on p. 2).

[3] Mark Peplow. «Robots rev up for Grand Challenge». In: *Nature* (Oct. 2005). ISSN: 1476-4687. DOI: 10.1038/news051003-9. URL: https://doi.org/10.1038/news051003-9 (cit. on p. 2).

[4] E. Blasch, A. Lakhotia, and G. Seetharaman. «Unmanned vehicles come of age: The DARPA grand challenge». In: *Computer* 39.12 (Dec. 2006), pp. 26–29. ISSN: 1558-0814. DOI: 10.1109/MC.2006.447 (cit. on p. 2).

[5] Murat Dikmen and Catherine Burns. «Trust in autonomous vehicles: The case of Tesla Autopilot and Summon». In: *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2017, pp. 1093–1098. DOI: 10.1109/SMC.2017.8122757 (cit. on p. 3).

[6] Sharon Poczter and Luka Jankovic. «The Google Car: Driving Toward A Better Future?» In: *Journal of Business Case Studies (JBCS)* 10 (Dec. 2013), p. 7. DOI: 10.19030/jbcs.v10i1.8324 (cit. on p. 3).

[7] Holger Caesar et al. *nuScenes: A multimodal dataset for autonomous driving*. 2020. arXiv: 1903.11027 [cs.LG] (cit. on p. 3).

[8] Adam Ziebinski, Rafal Cupek, Hueseyin Erdogan, and Sonja Waechter. «A Survey of ADAS Technologies for the Future Perspective of Sensor Fusion». In: *Computational Collective Intelligence*. Ed. by Ngoc Thanh Nguyen, Lazaros

Iliadis, Yannis Manolopoulos, and Bogdan Trawiński. Cham: Springer International Publishing, 2016, pp. 135–146. ISBN: 978-3-319-45246-3 (cit. on p. 3).

[9] Lars Svensson and Jenny Casey Eriksson. «Tuning for Ride Quality in Autonomous Vehicle : Application to Linear Quadratic Path Planning Algorithm». In: 2015 (cit. on p. 4).

[10] «Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles». In: (cit. on p. 4).

[11] Formula Student Germany. «Formula Student Rules 2020». In: 2020 (cit. on p. 7).

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on p. 12).

[13] Copeland B.J. *Artificial intelligence*. URL: https://www.britannica.com/technology/artificial-intelligence (cit. on p. 12).

[14] Expert.ai Team. *What is Machine Learning? A Definition*. URL: https://www.expert.ai/blog/machine-learning-definition/ (cit. on p. 13).

[15] Maurizio Di Paolo Emilio. *Intelligenza artificiale, deep learning e machine learning: quali sono le differenze?* URL: https://www.innovationpost.it/2018/02/14/intelligenza-artificiale-deep-learning-e-machine-learning-quali-sono-le-differenze/. (accessed: 20.03.2021) (cit. on p. 13).

[16] Artem Opperman. *Artificial Intelligence vs. Machine Learning vs. Deep Learning*. URL: https://towardsdatascience.com/artificial-intelligence-vs-machine-learning-vs-deep-learning-2210ba8cc4ac. (accessed: 26.03.2021) (cit. on p. 13).

[17] Rohit Gupta. *All about Deep Learning Tutorial*. URL: https://www.c-sharpcorner.com/article/deep-learning/ (cit. on p. 13).

[18] *CS231n Convolutional Neural Networks for Visual Recognition*. URL: https://cs231n.github.io/neural-networks-1/ (cit. on pp. 16, 17).

[19] Daniel Graupe. *Principles of Artificial Neural Networks*. 3rd. WORLD SCIENTIFIC, 2013. DOI: 10.1142/8868. eprint: https://www.worldscientific.com/doi/pdf/10.1142/8868. URL: https://www.worldscientific.com/doi/abs/10.1142/8868 (cit. on p. 15).

[20] *What is overfitting?* URL: https://www.ibm.com/cloud/learn/overfitting (cit. on pp. 24, 25).

[21] *Understand Local Receptive Fields In Convolutional Neural Networks*. URL: https://towardsdatascience.com/understand-local-receptive-fields-in-convolutional-neural-networks-f26d700be16c (cit. on pp. 27, 28).

[22] *Computer Vision*. URL: https://sites.google.com/site/mechatronicssdl/research-groups/computer-vision (cit. on p. 29).

[23] CreateBytes. *Real world Applications of computer vision*. URL: https://createbytes.com/insights/Real-world-Applications-of-computer-vision/ (cit. on p. 29).

[24] Reinhard Klette. *Concise Computer Vision - An Introduction into Theory and Algorithms*. Jan. 2014. ISBN: 978-1-4471-6319-0. DOI: 10.1007/978-1-4471-6320-6 (cit. on p. 29).

[25] Manivannan Murugavel. *Multiple Object Tracking Algorithms*. URL: https://manivannan-ai.medium.com/multiple-object-tracking-algorithms-a01973272e52 (cit. on p. 29).

[26] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. «Object detection with deep learning: A review». In: *IEEE transactions on neural networks and learning systems* 30.11 (2019), pp. 3212–3232 (cit. on p. 29).

[27] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. «SSD: Single Shot MultiBox Detector». In: *Lecture Notes in Computer Science* (2016), pp. 21–37. ISSN: 1611-3349. DOI: 10.1007/978-3-319-46448-0_2. URL: http://dx.doi.org/10.1007/978-3-319-46448-0_2 (cit. on p. 30).

[28] LaptrinhX. *Object Detection with SSD and MobileNet*. URL: https://laptrinhx.com/object-detection-with-ssd-and-mobilenet-3828024907/ (cit. on pp. 31, 33).

[29] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. «Mobilenets: Efficient convolutional neural networks for mobile vision applications». In: *arXiv preprint arXiv:1704.04861* (2017) (cit. on pp. 32, 33).

[30] Gaurav Maindola. *A Brief History of YOLO Object Detection Models From YOLOv1 to YOLOv5*. URL: https://machinelearningknowledge.ai/a-brief-history-of-yolo-object-detection-models/ (cit. on p. 34).

[31] Pierrick RUGERY. *Explanation of YOLO V4 a one stage detector*. URL: https://becominghuman.ai/explaining-yolov4-a-one-stage-detector-cdac0826cbd7 (cit. on p. 34).

[32] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*. 2016. arXiv: 1506.02640 [cs.CV] (cit. on p. 35).

[33] Joseph Redmon and Ali Farhadi. «YOLO9000: better, faster, stronger». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271 (cit. on p. 37).

[34] Joseph Redmon and Ali Farhadi. «Yolov3: An incremental improvement». In: *arXiv preprint arXiv:1804.02767* (2018) (cit. on p. 38).

[35] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. arXiv: 2004.10934 [cs.CV] (cit. on p. 40).

[36] Vision team. *What is a stereo vision camera?* URL: https://www.e-consystems.com/blog/camera/what-is-a-stereo-vision-camera/ (cit. on p. 44).

[37] StereoLabs. *StereoLabs*. URL: https://www.stereolabs.com/docs/ (cit. on p. 47).

[38] 2021 Fight's On. *Inertial navigation system (INS)*. URL: https://fightson.net/inertial-navigation-system-ins/ (cit. on p. 50).

[39] Australian Academy of Science. *What is the Global Navigation Satellite System?* URL: https://www.science.org.au/curious/technology-future/what-global-navigation-satellite-system (cit. on p. 51).

[40] Stefano Feraco, Sara Luciani, Angelo Bonfitto, Nicola Amati, and Andrea Tonoli. «A local trajectory planning and control method for autonomous vehicles based on the RRT algorithm». In: *2020 AEIT International Conference of Electrical and Electronic Technologies for Automotive (AEIT AUTOMOTIVE)*. 2020, pp. 1–6. DOI: 10.23919/AEITAUTOMOTIVE50086.2020.9307439 (cit. on pp. 52, 53).

[41] Stefano Feraco, Angelo Bonfitto, N. Amati, and Andrea Tonoli. «Redundant Multi-Object Detection for Autonomous Vehicles in Structured Environments». In: *Komunikacie* 24 (Jan. 2022), pp. C1–C17. DOI: 10.26552/com.C.2022.1.C1-C17 (cit. on p. 53).

[42] Jorge Fuentes-Pacheco, José Ruiz-Ascencio, and Juan Manuel Rendón-Mancha. «Visual simultaneous localization and mapping: a survey». In: *Artificial intelligence review* 43.1 (2015), pp. 55–81 (cit. on p. 56).

[43] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. «ROS: an open-source Robot Operating System». In: vol. 3. Jan. 2009 (cit. on p. 56).

[44] ROS.org. *Nodes*. URL: http://wiki.ros.org/Nodes (cit. on p. 57).

[45] Erle Robotics GitBook. *ROS: Concepts*. URL: https://erlerobot.github.io/erle_gitbook/en/ros/ROS-concepts.html (cit. on pp. 57, 58).

[46] Wikipedia contributors. *Robot Operating System — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Robot_Operating_System&oldid=1045620915. [Online; accessed 2-October-2021]. 2021 (cit. on p. 58).

[47] Rasines, Javier and Khoche, Ajinkya. *Perception and SLAM for Formula Student Driverless- 17/18*. https://github.com/javirrs/PerceptionAndSlam_KTHFSDV1718. 2018 (cit. on pp. 59, 68, 72).

[48] Harvey B Mitchell. *Multi-sensor data fusion: an introduction*. Springer Science & Business Media, 2007 (cit. on p. 60).

[49] Peter S Maybeck. *Stochastic models, estimation, and control*. Academic press, 1982 (cit. on p. 61).

[50] Analytics Vidhya. *Kalman Filters: A step by step implementation guide in python*. URL: https://medium.com/analytics-vidhya/kalman-filters-a-step-by-step-implementation-guide-in-python-91e7e123b968 (cit. on p. 61).

[51] Eric A Wan and Rudolph Van Der Merwe. «The unscented Kalman filter for nonlinear estimation». In: *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No. 00EX373)*. Ieee. 2000, pp. 153–158 (cit. on p. 62).

[52] Josep Aulinas, Yvan Petillot, Joaquim Salvi, and Xavier Lladó. «The SLAM problem: a survey». In: *Artificial Intelligence Research and Development* (2008), pp. 363–371 (cit. on p. 62).

[53] TensorFlow Blog. *Predicting Known Unknowns with TensorFlow Probability — Industrial AI, Part 2*. URL: https://blog.tensorflow.org/2018/12/predicting-known-unknowns-with-tensorflow-probability-part2.html (cit. on p. 62).

[54] Favelli Stefano. *Robust Localization for an Autonomous Racing Vehicle*. Master thesis degree. 2021 (cit. on p. 63).

[55] ROS.org. *sbg_driver*. URL: http://wiki.ros.org/sbg%5C_driver (cit. on p. 63).

[56] Mohinder S Grewal, Lawrence R Weill, and Angus P Andrews. *Global positioning systems, inertial navigation, and integration*. John Wiley & Sons, 2007 (cit. on p. 64).

[57] ROS.org. *robot_localization*. URL: http://wiki.ros.org/robot%5C_localization (cit. on p. 64).

[58] ROS.org. *robot_localization wiki*. URL: http://docs.ros.org/en/melodic/api/robot_localization/html/index.html (cit. on p. 64).

[59] ECET 49900/58100. *TF (transform) in ROS*. URL: https://web.ics.purdue.edu/~rvoyles/Classes/ROSprogramming/Lectures/TF%5C%20(transform)%5C%20in%5C%20ROS.pdf (cit. on p. 64).

[60] EMLID. *Reach M+*. URL: https://store.emlid.com/product/reachm-plus/ (cit. on p. 65).

[61] Nikhil Gosala et al. «Redundant perception and state estimation for reliable autonomous racing». In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 6561–6567 (cit. on p. 66).

[62] Luis E. Ortiz, Elizabeth V. Cabrera, and Luiz Marcos Garcia Gonçalves. «Depth Data Error Modeling of the ZED 3D Vision Sensor from Stereolabs». In: *Electronic Letters on Computer Vision and Image Analysis* 17 (2018), pp. 1–15 (cit. on p. 71).