FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Machine Learning Based Controller for the Robot used in Autonomous Driving Competition

Gonçalo Freitas Ferreira Martins

Integrated Master's in Electrical and Computers Engineering

Supervisor: Armando Jorge Sousa (Ph.D.)

Co-Supervisor: Válter Costa (Ph.D.)

July 30, 2021

Abstract

In the last few years, the application of algorithms based on Artificial Intelligence and Machine Learning (ML) has grown significantly in many areas, including autonomous driving. Autonomous driving has attracted the interest of the automotive industry, and of many researchers within the scientific community. This is reflected in the emergence of numerous events around the world promoting research and development in this area in the form of competitions. One of those competitions is the Autonomous Driving Competition (ADC) of the Portuguese Robotics Open (PRO). This competition presents a set of challenges for autonomous mobile robots on a track very similar to a conventional road.

The project Major Alvega consists of a realistic simulation environment of the competition scenario and an Ackermann steering robot. This robot participated in the 2019 edition of the ADC and it uses computer vision, control theory, and odometry based methods to control its navigation. These methods are complex to process and integrate. To make the robot control system simpler and obtain better performance, ML methods based on Artificial Neural Networks (ANNs) and Supervised Learning algorithms were applied. It was intended, with those algorithms, to provide the robot the capacity of predicting a steering angle that would keep it within the track in and end-to-end manner, using only images captured by a camera. For this purpose, two state-of-the-art ANNs for autonomous driving were used: PilotNet and JNet. To build the training dataset, previous data of the robot driving on the track in a real competition environment was employed. In cases where this data was insufficient, the simulation environment from Major Alvega's project was used to generate more data.

Experiments performed on the track in a simulation environment have shown that the robot was able to successfully travel along the different possible routes at different speeds, while staying within the lanes and with low deflections from the ideal trajectory. It was also observed a more steady and smoother steering in comparison with the previous navigation system. Moreover, with the integration of this new controller in Major Alvega's previous architecture, the robot was also able to circumvent obstacles. The results of these experiments revealed that it is possible to effectively control the robot's navigation using an end-to-end learning framework, while significantly reducing the number of intermediate processing steps.

Resumo

Nos últimos anos, tem-se assistido à aplicação de algoritmos baseados em Inteligência Artificial e Aprendizagem Automática em inúmeras áreas, entre as quais a condução autónoma. Esta área tem cativado o interesse da indústria automóvel e de muitos investigadores dentro da comunidade científica. Tal reflete-se na criação de inúmeros eventos a nível mundial que promovem a investigação e o desenvolvimento nesta área sob o formato de competições. Uma dessas competições é a competição de Condução Autónoma do Festival Nacional de Robótica. Esta competição apresenta um conjunto de desafios para robôs autónomos numa pista muito semelhante a uma estrada convencional.

O projeto Major Alvega consiste num ambiente de simulação realista do cenário da competição e num robô de locomoção Ackermann que participou na edição de 2019 da Competição de Condução Autónoma. Este robot utiliza técnicas baseadas em visão computacional, teoria do controlo e odometria para o controlo da sua navegação. Estes métodos são bastante complexos e difíceis de integrar. De forma a tornar o sistema de controlo de navegação do robot mais simples e obter um melhor desempenho, foram aplicados métodos de aprendizagem automática baseados em algoritmos de redes neuronais e aprendizagem supervisionada. Utilizando estes algoritmos, pretendeu-se que o robot aprendesse de modo *end-to-end* a determinar um ângulo de volante que o mantivesse dentro da pista, recorrendo apenas a imagens captadas por uma camera. Para o efeito, foram utilizadas duas redes neuronais estado-da-arte para condução autónoma: a PilotNet e a JNet. Para formar os dados para treinar estas redes, utilizaram-se dados prévios do robot a conduzir na pista em ambiente real da competição. Em situações que estes dados eram insuficientes, foi utilizado ambiente de simulação do projecto do Major Alvega para criar mais dados.

Avaliações feitas na pista em ambiente de simulação, demonstraram que o robot, utilizando o controlador proposto, foi capaz de percorrer com sucesso os diferentes percursos possíveis e a diferentes velocidades, mantendo-se dentro das faixas e com baixos desvios relativamente à trajetória ideal. Foi também observado uma condução mais estável e suave em comparação com o sistema de navegação anterior. Além disso, com a integração deste novo controlador na arquitetura do Major Alvega, o robot também foi capaz de contornar obstáculos. Os resultados destas avaliações revelaram que é possível controlar eficazmente a navegação do robot usando aprendizagem *end-to-end*, reduzindo-se significativamente o número de passos intermédios de processamento.

Agradecimentos

Antes de mais, quero agradecer aos meus orientadores, Armando Sousa e Valter Costa, por todo o apoio e orientação constante que me deram ao longo desta dissertação e por terem confiado em mim para fazer parte do seu projecto.
Não podia também deixar de agradecer à minha família. Aos meus pais, por toda a educação que me deram ao longo destes anos. Estiveram sempre lá nos bons e nos maus momentos da minha vida e apoiaram-me sempre. São, sem sombra de dúvidas, as pessoas que mais me ajudaram a acreditar em mim e a ensinar-me o que se leva desta vida. Também quero agraceder ao meu irmão, que tanto adoro e que tanto me apoiou, e à minha avó, a minha maior fã. Este trabalho é dedicado a eles.
Aos meus amigos, não só por me terem acompanhado este tempo todo na biblioteca da FEUP, mas também por me terem proporcionado os melhores momentos da minha vida. Agradeço toda a paciência que tiveram comigo e por me ajudarem nos momentos mais difíceis.

Gonçalo Martins

"I don't know where I'm going,
but I'm on my way."

Carl Sagan

Contents

List of Figures

List of Tables

Abbreviations

ADC	Autonomous Driving Competition
AI	Artificial Intelligence
ALVINN	Autonomous Land Vehicle in a Neural Network
ANN	Artificial Neural Network
API	Application Programming Interface
BGD	Batch Gradient Descent
CNN	Convolutional Neural Network
C-LSTM	Convolutional Long Short-Term Memory Recurrent Neural Network
CPU	Central Process Unit
DARPA	Defense Advanced Research Projects Agency
DNN	Deep Neural Networks
ELU	Exponential Linear Unit
GPS	Global Positioning System
GPU	Graphics Processing Unit
GRU	Gated Recurrent Units
GUI	Graphical User Interface
IL	Inner Lane
JPEG	Joint Photographic Experts Group
LIDAR	Light Detection And Ranging
LPF	Low Pass Filter
LSTM	Long Short Term Memory
MAE	Mean Absolute Value
MBGD	Mini-Batch Gradient Descent
MCE	Mean Continuity Error
ML	Machine Learning
MSE	Exponential Linear Unit
OL	Outer Lane
PDS	Previously Developed System
PID	Proportional–Integral–Derivative
PNG	Portable Network Graphics
PRO	Portuguese Robotics Open
RL	Reinforcement Learning

RMSE	Root Mean Squared Error
RNN	Recurrent Neural Networks
ROS	Robotic Operating System
ReLU	Rectified Linear Unit
SDF	Simulation Description Format
SGD	Stochastic Gradient Descent
SL	Supervised Learning
SVG	Scalable Vector Graphics
TF	TensorFlow
TPU	Tensor Processing Unit
UL	Unsupervised Learning
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Context

Over the last few years there has been an increase in the application of algorithms based on Artificial Intelligence (AI) and Machine Learning (ML) in many areas. This is a result of the growth in both the processing power of computers and the data availability. These algorithms are able to learn to perform several tasks automatically using only data provided by humans. In the last decade, there are examples of cars that make use of deep learning based technologies, such as Tesla[1] and Waymo[2]. The application of ML proves to be the most promising to guarantee a maximum level of autonomy in cars, and ultimately resolve the driving paradigm. This would lead to greater mobility for both the elderly and disabled people, less traffic congestion with less air pollution associated, reduced energy consumption, less material damages and, more importantly, fewer human deaths on the road. The area of autonomous driving has also captured the interest of researchers within the scientific community. This is denoted by the organization of numerous events around the world that promoting research and development in this area. These events are usually in the form of competitions. Examples of such competitions include: the DARPA Grand Challenge [1] and the Autonomous Driving Competition (ADC) [2] of the Portuguese Robotics Open (PRO) [3]. The latter occurs annually in Portugal and presents a set of challenges for autonomous mobile robots on a track very similar to a conventional road. The Major Alvega [4] project consists of a small steering Ackermann robot that participated in the 2019 edition of this same competition. This project also included the creation of a realistic simulation environment [5] of the competition scenario.

[1] https://www.tesla.com/. [Last accessed: 22/06/2021]
[2] https://waymo.com/. [Last accessed: 22/06/2021]

1.2 Objectives

One of the Major Alvega's most important system is the one that controls its navigation. This system uses computer vision, control theory, and odometry localization based techniques to determine the steering angle. In the current system of the Major Alvega, these methods are hard-coded, complex and difficult to optimise. To overcome this, and following the trend of the recent autonomous cars driving algorithms, the main goal of this dissertation is to implement ML algorithms for the robot driving controller to make it simpler and to achieve better performance. It is pretended that this new controller is able to calculate a steering angle that keeps the robot within the track in the respective lanes. For the development of the algorithms and further tests, the capabilities of the simulation environment developed within Major Alvega's project will be used. In summary, the main objectives of this dissertation are the following:

- To study ML-based algorithms to develop a new controller for the system of the Major Alvega robot;
- The new ML-based controller should be able to predict, in an end-to-end manner, the appropriate steering angle that keeps the robot within the track and its respective lanes based on images captured from a front-facing camera;
- To integrate the new ML-based controller into Major Alvega's system to perform the different ADC challenges;
- To conduct a comparative analysis of Major Alvega's performance with the new and the old navigation systems.

1.3 Document Structure

This document is structured as follows: Chapter 2 introduces Major Alvega's project and the ADC, the competition in which this project participates. It also describes the simulation environment and the previous architecture of Major Alvega that was used when the robot participated in the 2019 edition of the ADC. Chapter 3 includes the literature review and a state-of-the-art survey regarding deep learning and its application in autonomous driving. Chapter 4 contains the description of the proposed ML-based controller and the methodologies followed. Chapter 5 presents the results of the proposed controller in driving the robot on the simulation environment track, including also a section of discussion. Finally, Chapter 6 draws the main conclusions and points out future work.

Chapter 2

Major Alvega Project and ADC

This chapter introduces the project of Major Alvega that participated in the Autonomous Driving Competition (ADC) of the Portuguese Robotics Open (PRO) of 2019. In Section 2.1 the ADC competition is introduced, as well as the 3 different challenges that compose it. The simulation environment used for the development of this dissertation is presented in Section 2.2, such as the tools used for that purpose: the Robotic Operating System (ROS) (Section 2.2.1) and the Gazebo simulator (Section 2.2.2). Finally, in Section 2.3, a brief description of the ROS system architecture implemented in the robot Major Alvega is given.

2.1 Autonomous Driving Competition of the Portuguese Robotics Open

The PRO is an event promoted by the Portuguese Robotics Society (SPR – Sociedade Portuguesa de Robótica) that aims to promote science and technology among youngsters, teachers, researchers and the general public, through autonomous robot competitions [3]. One of these competitions is the ADC that presents a variety of challenges for fully autonomous mobile robots that must traverse a route along a closed track with strong similarities to a conventional road. Some of these challenges, defined in the rules of the competition [6], try to replicate the real challenges of self driving cars, such as driving alongside the track, detecting traffic lights and signs, avoiding obstacles, performing different parking situations and circumventing work zones defined by small traffic cones.

The competition track (Figure 2.1) has a total size of $16.7 \times 6.95\ m^2$, it is surrounded by two parallel continuous side lines and contains two lanes separated by one single broken line. Moreover, it is composed of two circular curvatures and one straight leg including a zebra crossing placed in its center. In addition, it contains two parking areas: one parallel parking band, located at the right of the lane after the zebra crossing and one bay parking zone, with two parking spots, highlighted with the letter “P” and placed outside of the driving track. At the zebra crossing area

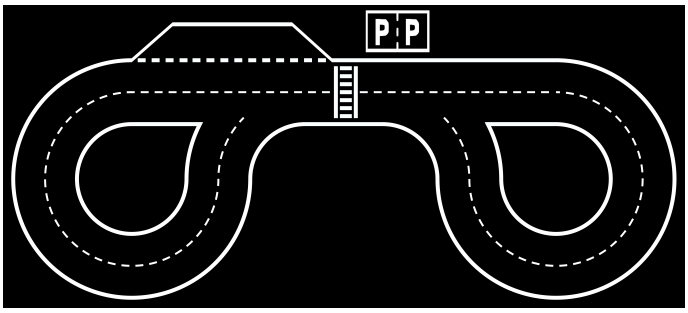


Figure 2.1: Competition track of the ADC's 2019 edition.

it is mounted a signalling panel (Figure 2.2) responsible for indicating orders and directions that the robot must obey (Table 2.1).

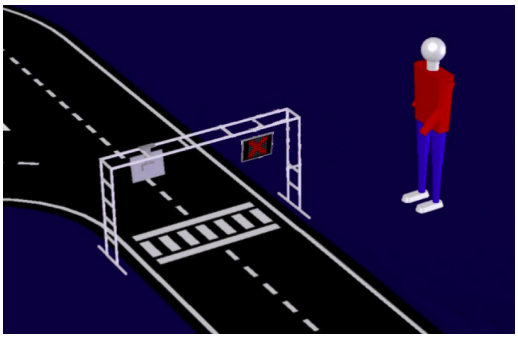


Figure 2.2: Signaling panel [6].

Table 2.1: Signaling panels of the ADC. Adapted from: [6].

Action	Signal
Follow to the left	
Follow to the right	
Follow straight ahead	
Stop	
End of trial	
Follow to parking area	

According to the rules from the 2019 edition [6], there are 3 types of challenges: The driving challenge, the parking challenge and the vertical traffic sign detection challenge.

2.1.1 Driving Challenge

The unanimous goal of the driving challenges is to complete two laps around the circuit, starting from the departure/arrival zone next to the zebra crossing and returning to the same area. This has to be made in the shortest amount of time and with the fewest penalties possible. The four different driving challenges with an incremental level of difficulty are:

- **Driving at pure speed challenge (D1)**: The signalling panel is only used to trigger the robot departure;
- **Driving with signs challenge (D2)**: The robot must drive along the track following the indications given by the signaling panel;
- **Driving with signs and obstacles challenge (D3)**: Same as the D2 challenge with the addition of having to dodge two obstacles placed in the second half of the track. These obstacles are represented always by a green box with 60 cm^2 base and 20 cm as minimum height;
- **Driving with all challenge (D4)**: Same as challenge D3 with the addition of a working zone traced by orange and white traffic cones.

2.1.2 Parking Challenge

The unanimous goal of the parking challenges is to park the robot in the respective parking areas triggered by the ”Follow to parking area” signal indicated by the signaling panel. As mentioned before, there are two different parking areas, the parallel and the bay parking. Concerning the challenges, they are divided in the following way:

- **Parallel parking without obstacles (P1):** After the parking signal, the robot must stop in the parking band, after the zebra crosswalk, parallel to the driving lane .
- **Parallel parking with obstacles (P2):** Same as above, but two obstacles are placed in the parking band without a specific position. However, the distance between them must not be less than twice the length of the performing robot.
- **Bay parking without obstacles (B1):** After the parking signal, the robot must stop in one of the two spots in the bay parking zone.
- **Bay parking with obstacles (B2):** Same as above, but one obstacle box is placed in one of the parking spots, forcing the robot to park in the other spot.

2.1.3 Vertical Traffic Signs Detection Challenge

The goal of the vertical signs detection challenge (V1) is to identify and detect the traffic signs shown in the Figure 2.3 while driving on the track. Six of them (two per type) are placed next to the circuit.

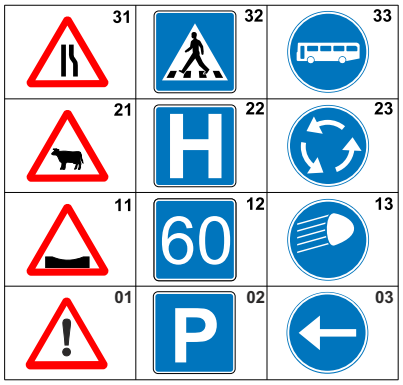


Figure 2.3: Vertical traffic signs used in challenge V1 [6].

The details about the scores and the penalties as well as more detailed information can be found in the rules of the ADC's 2019 Edition [6].

2.2 Simulation Environment

A simulator plays an important role in the development of applications in robotics. Through a simulator it is possible to develop virtual models of physical systems and their entire surrounding environment, allowing a reproduction of real situations in a virtual environment with a satisfactory degree of similarity. Hence, it is possible to easily perform a large number of tests with different initial conditions and/or in different scenarios. In addition, there are other advantages in using a simulated environment, such as prototyping a robot, testing different algorithms without concern for damages at the hardware level, ease of debugging, creation of datasets and portability. The last one is one of the most valuable advantages, since replicating the conditions in which a robot is inserted or even carrying it to a certain position can be unfeasible, costly and time-consuming.

In this context, previous works of [7, 8, 4, 5] consisted of the creation of a PRO ADC simulator. The simulator results had a maximum relative error of 4.65% in the odometry tests and led to an accuracy of 99.37% with the dataset used for traffic signals recognition tests [8]. All this leads to the conclusion that the developed simulator is extremely realistic with only a small gap between simulation and reality.

The simulator was developed using ROS and Gazebo and represents two robot models that participated in previous editions of the competition: Conde [7, 8], a differential locomotion robot and Major Alvega [4], an Ackermann locomotion robot. Moreover, it also supports every challenge of the competition mentioned before in Section 2.1. The simulator is open source and is available for download in [5].

2.2.1 Robotic Operating System

The Robotic Operating System (ROS)[1] [9] is an open-source framework for developing applications in robotics widely used in this field. It provides several features, such as hardware abstraction, low-level device control, message-passing between different processes as well as package management. In addition, it offers several libraries, tools and standardized messages with solutions to frequent problems in robotic applications. The main languages of code development are Python and C++.

The communication between the different processes, denominated by ROS nodes, is done through topics following the publisher-subscriber topology. In ROS, nodes publish messages on topics and the other nodes can subscribe to them and read the stored messages. ROS also allows communication between nodes through services according to the client/server topology. A service is a specific task performed by a node dedicated exclusively to that task. Therefore, the interaction between client and server works by exchanging two messages: one with the request and one with the reply.

These two types of communication are different in the number of nodes that can communicate with each other. While topics allow many-to-many communication, services only allow one-to-one communication. Both types of communication are managed by the ROS Master. The ROS Master provides naming and registration services to the nodes in the ROS system. It tracks not only the publishers and subscribers to the topics but also to the services. One of its main responsibilities is to locate the ROS nodes and establish communication channels between them through a protocol.

Figure 2.4 depicts, in a holistic view, the simplified architecture of a ROS system as well as the two types of existing communications.

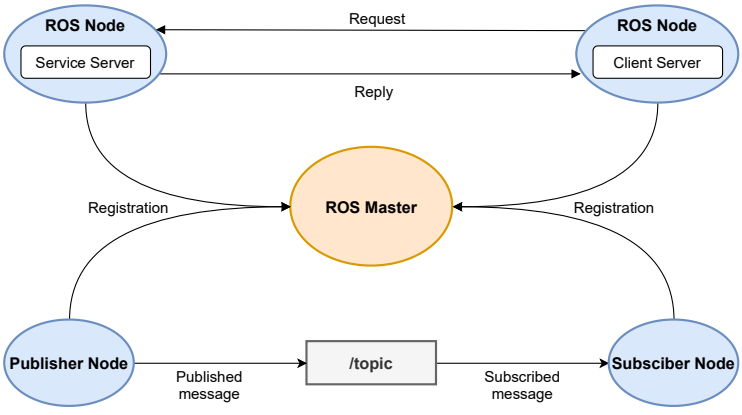


Figure 2.4: ROS example of the client/service and the publisher/subscriber topologies.

The ROS environment also possesses visualization tools and other tools that help in testing and debugging the system. These include system graph visualization tools (`rqt_graph`), 3D visualization (`rviz`), logging tools (`rqt_console`), recording and (`rosbag` and `rqt_bag`) and a set of command-line tools for testing and debugging various parts of the system such as, nodes

[1] https://www.ros.org/. [Last accessed in 21/05/2021]

(`rosnode`), topics and their messages (`rostopic`), services (`rosservice`), system parameters (`rosparam`), among others.

Among these tools, one should highlight the `rqt_graph` that allows the visualization of the architecture of the system that is being executed, i.e., the connection between the nodes and the topics and the information in the messages being sent. Also, the `rosbag` allows the recording and then playback of messages on certain topics and still handles the data collected for other purposes, such as creating datasets. The `rqt_bag` not only allows the recording and playback of messages, but it also makes it possible to visualize them. Finally, the various command-line tools enable the user to see the elements of the system, their type, the messages transmitted and the information about them.

In this manner the ROS presents a distributed and modular architecture that has the great advantage of being able to distribute the nodes between different computers or robots.

2.2.2 Gazebo

The Gazebo[2] is an open source 3D simulator used for robotic applications and is the default simulator of ROS. It is capable of simulating object models, including robots, present in its environment. Objects are treated as a set of rigid links interconnected by joints. Sensors can also be associated with each link.

The Gazebo system is divided into two parts, the server, `gzserver`, and the client, `gzclient`. On the one hand, the server is responsible for calculating the physics, the rendering of the objects and the information captured by the sensors. On the other hand, the client is responsible for providing a Graphical User Interface (GUI) that allows visualization and interaction with the simulation environment.

The physics default engine is the Open Dynamics Engine (ODE)[3]. However, it is important to note its capability of using others, such as the Bullet[4], the Simbody[5] and the Dynamic Animation and Robotics Toolkit (DART)[6]. The physics engine is responsible for simulating the dynamics and kinematics of objects belonging to the simulation environment. The characteristics provided by this engine are collision detection, types of movement made by the joints, mass and inertia.

When it comes to graphics, the Gazebo simulator uses an existing open-source graphics engine, the Object-Oriented Graphics Rendering Engine (OGRE)[7]. This engine is used to render the simulated 3D scenarios. These scenarios allow the user to present a GUI with the state of the world. Rendering includes simulation of lighting, textures and the sky.

Gazebo provides a library with several categories of sensors such as altimeters, cameras, contact sensors, depth, GPS, sonar, among others. Furthermore, it is also possible to create new

[2] `http://gazebosim.org/`. [Last accessed in 21/05/2021]
[3] `http://www.ode.org/`. [Last accessed in 21/05/2021]
[4] `https://pybullet.org/wordpress/`. [Last accessed in 21/05/2021]
[5] `https://simtk.org/projects/simbody/`. [Last accessed in 21/05/2021]
[6] `https://dartsim.github.io/`. [Last accessed in 21/05/2021]
[7] `https://www.ogre3d.org/`. [Last accessed in 21/05/2021]

sensors using plugins. It is by the information given by these sensors that the robot model perceives the virtual environment, making the extraction of information even more realistic in the simulator by adding noise to the sensors.

The objects present in the simulation environment are represented in Simulation Description Format (SDF) format, an Extensible Markup Language (XML) dialect, which describes the shape and textures of its links, joints, sensors and model plugins. The representation of the object's links includes attributes such as shape, textures, luminosity and collision and inertia models. The joints are represented with properties like the joint's type, the links it connects, the degrees of freedom and movement restrictions. The world, on the other hand, can be seen as an object containing other objects, lighting elements, world plugins and global properties.

It is also worth mentioning that Gazebo allows extending its functionalities through the addition of plugins. These plugins can refer to the world, the model or its visual part, the sensors, the graphical interface and even the Gazebo system itself.

Finally, the GUI of the Gazebo (Figure 2.5) is an interface that allows the user not only to visualize the simulation environment and the behavior of the models but also their manipulation. The spectrum of manipulations ranges from adding and removing objects to translate and to rotate them, changing the lighting and the simulation speed.

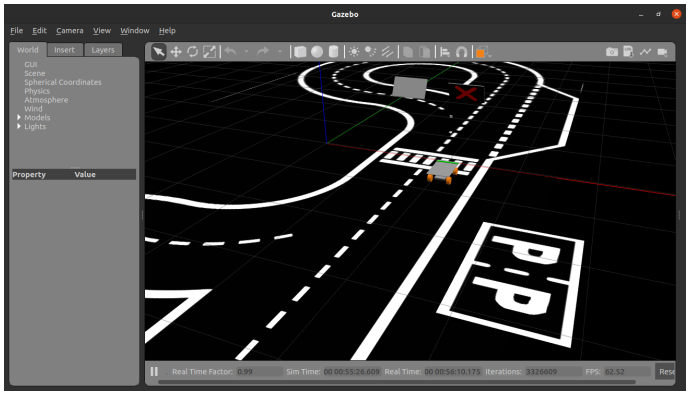


Figure 2.5: Gazebo GUI.

2.2.2.1 Design of the Robot World

The modeling of the track corresponds to the definition of a texture that represents the track and its application as a visual representation of a model. The model is treated as a static element with the visual representation and the collision model defined as a plane with the appropriate dimensions. The application of the texture to the plane must take into account the resolution of the image and the actual dimensions of the track. The design is made through a SDF configuration file, where the various parameters relating to the components of a material are specified.

In a first approach, the texture of the robot world floor was applied using a Portable Network Graphics (PNG) file that resulted from an extraction of the image of the track present in the ADC

rules document [6]. However, it was later noticed that some dimensions of the track were not exactly accomplished in the simulator due to the occurrence of distortion phenomena. To eliminate these phenomena, in a second approach, the ADC track used as texture was designed in Scalable Vector Graphics (SVG) format using the Inkscape[8] software. This way, it was ensured that the track used for further tests had the exact dimensions according to the competition rules and that there were no distortion phenomena when exporting the image to the PNG format and using it as a floor texture. The bay parking area was designed in a similar way, both being designed in flat ground and well-defined light conditions. Nevertheless, the Gazebo simulator offers the possibility of adding spotlights or changing lighting parameters as well as a model with uneven ground.

In the zebra crosswalk zone, two monitors were added, consisting of two respective links in order to display the semaphores. These semaphores use another plugin from ROS called `libgazebo_ros_video` and an additional node was made to control the displays on the simulator [7]. This node has a menu which allows the user to select the semaphore to be displayed on both monitors according to Table 2.1.

For challenge V1 (Section 2.1.3), similar to the signalling panel, 12 monitors were added, consisting of 12 links along the track. With the aid of the same plugin it is possible to display the 12 traffic signs of the competition according to the Figure 2.3. For the challenges with obstacles, the boxes used as obstacles were modeled by a green parallelepiped with no texture. The working area consists of 9 traffic cones that can be arranged on the track by the user.

One can conclude that the developed simulator is capable of covering all challenges of ADC's PRO and Figure 2.6 depicts, in a holistic view, the capabilities of the simulator discussed in this subsection.

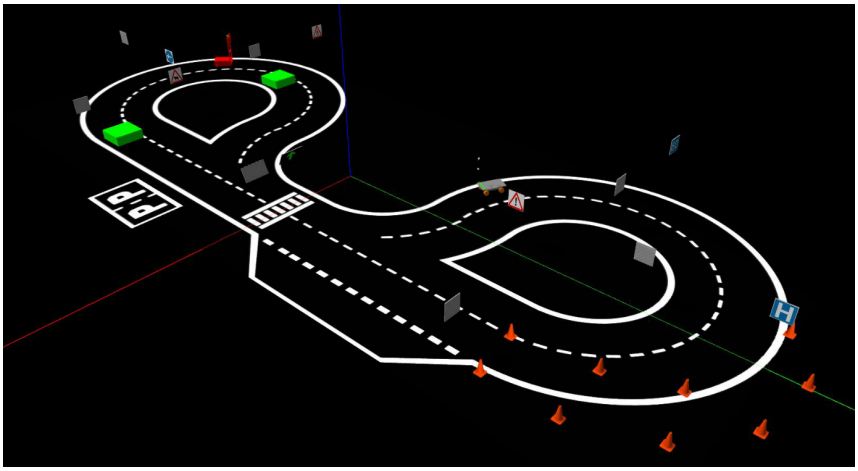


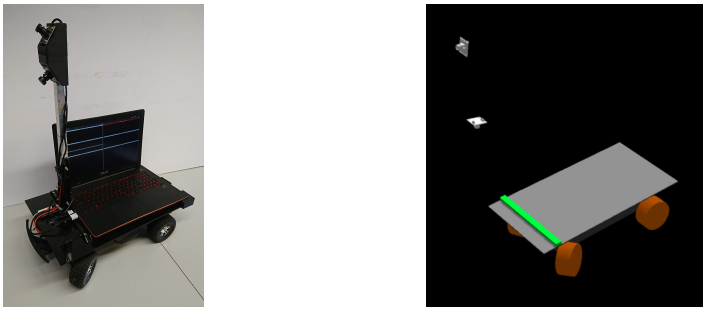
Figure 2.6: Simulation world of the ADC [4].

[8] `https://inkscape.org/`. [Last accessed in 21/05/2021]

2.2.2.2 Design of the Robot Model

As mentioned previously, the simulator is capable of representing two robot models: Major Alvega, the robot that participated in the 2019 edition of ADC, and the Conde, which participated in the 2018 and 2017 editions. In the scope of this dissertation, only the Major Alvega robot was used, therefore the focus will be directed to this robot.

The Major Alvega (Figure 2.7) is a 30×50 cm^2 size driving robot that uses an Ackermann steering mechanism, the same as a standard real-life car. In addition, the robot is equipped with a vision system composed of 2 cameras: one pointing upwards to detect signaling panels and traffic signs and another pointing downwards to detect the track and the lines that comprise it. These cameras have fisheye lenses in order to increase their field of view, making it possible to view both lanes. Figure 2.7 shows the real robot and its representation in the virtual environment.



(a) Major Alvega real robot (b) Major Alvega simulated model

Figure 2.7: Major Alvega Robot in reality and in simulation environment [4].

Hence, Major Alvega is composed of 3 parts (wheels, chassis, cameras) that were designed using a CAD software and exported to a Unified Robot Description Format (URDF) file, which is an XML file for representing robot models. Then, this URDF file was converted to Xacro, an XML macro language, which allows more modularity and code reuse when defining a URDF model.

The first part that was designed was the chassis. This step was done by measuring the actual dimensions of the real robot and applying them to the Xacro configuration file. The chassis was made of two parallelepipeds, an upper and a lower one that were both connected through a bar, thus composing Major Alvega's base link.

The next step was the design of the locomotion system formed by the wheels. The ackermann locomotion robot consists of two front directional wheels and two rear wheels with a motor that is responsible for moving the robot. All these elements were connected and linked to the chassis.

Finally, the last step was the integration of the 2 cameras. Both cameras were added to the robot model by using a sensor tag of type *wideanglecamera*. In order to connect the cameras to ROS `libgazebo_ros_camera` was used. Using this plugin it is possible to define camera frame rate, resolution, intrinsic camera parameters, and the topic on which the camera will publish the captured image. Then the cameras were also linked to the chassis to complete the design of Major Alvega.

2.3 ROS Architecture

Major Alvega's control system was divided into several subsystems where each one is a node implemented in ROS. Figure 2.8 demonstrates the simplified ROS architecture of the robot, where the nodes are represented by the red oval blocks and the topics are represented by the blue rectangular blocks. Major Alvega participated in the 2019 edition of the ADC using this architecture and the respective developed software.

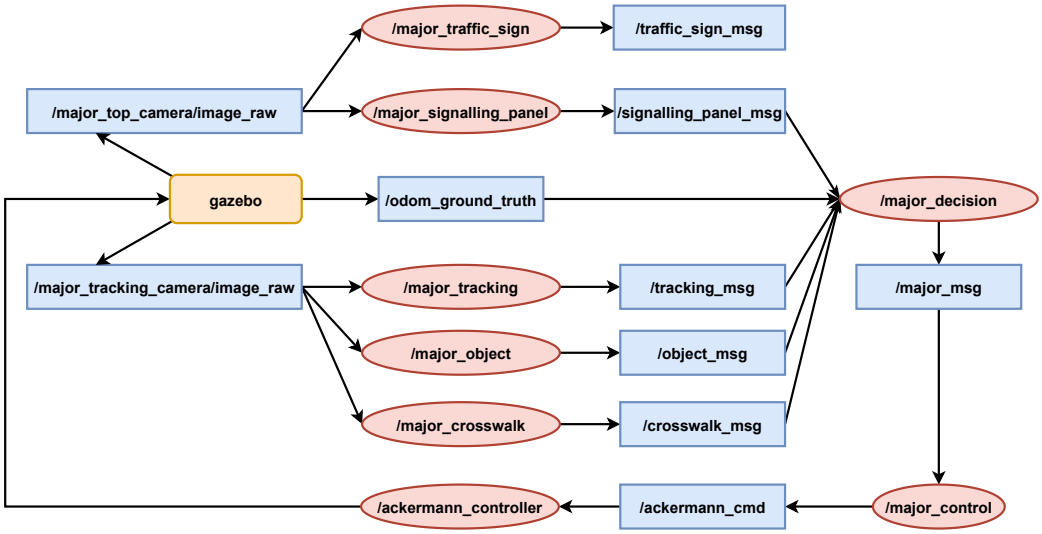


Figure 2.8: Major Alvega ROS architecture system.

The topics `/major_tracking_camera/image_raw` and `/major_top_camera/image_raw` are where the images are published, these being obtained from the camera pointing downwards (tracking camera) and the camera pointing upwards (top camera), respectively. The information regarding the odometry of the robot is published in the topic `/odom_ground_truth`, more precisely the robot pose in the simulation environment on Gazebo. Based on the information published in these topics, the other nodes integrating the Major Alvega system have the following functions:

- **`major_signaling_panel` node:** This node is responsible for detecting and identifying the indicating signals in Table 2.1 displayed on the signaling panels. The identification of the detected signals is published in the topic `/signalling_panel_msg`.

- **`major_traffic_sign` node:** This node is responsible for detecting and identifying the vertical traffic signs in Figure 2.3. The identification of the detected traffic sign is published in the topic `/traffic_sign_msg`.

- **`major_object` node:** This node is responsible for detecting objects used as obstacles. The information that an obstacle has been detected is published in the topic `/object_msg`.

- **`major_crosswalk` node:** This node is responsible for detecting the zebra crosswalk. The information that the zebra crosswalk was detected is published in the topic `/crosswalk_msg`.
- **`major_tracking` node:** This node is responsible for tracking the lines of the lane and calculate the distance and angle of the robot's referencial in relation to them. That information is then published in the `/tracking_msg` topic.
- **`major_decision` node**: This node owns the robot's intelligence. It is responsible for gathering the information from the cameras processed by the other nodes and the odometry information and for publishing distance, angle and speed references to be used by the control node. These references are published in the topic `/major_msg`.
- **`major_control` node:** This node is responsible for calculating and updating Major Alvega's velocity and steering angle using the references published in the topic `/major_msg` and applying a PID controller. This information is published in the `/ackermann_cmd` topic, which updates the robot simulation in the Gazebo world via `/ackermann_controller`.

The algorithms for the tracking node and for the detection of signaling panels are described in [10, 11] at the time applied to the Conde robot and later also applied in the Major Alvega.

Chapter 3

Literature Review and State of the Art

In this chapter, Section 3.1 reviews the literature, presenting the fundamental knowledge and background required for this dissertation. Moreover, in Section 3.2 several state of the art architectures with related works associated are also mentioned.

3.1 Fundamentals

3.1.1 Machine Learning

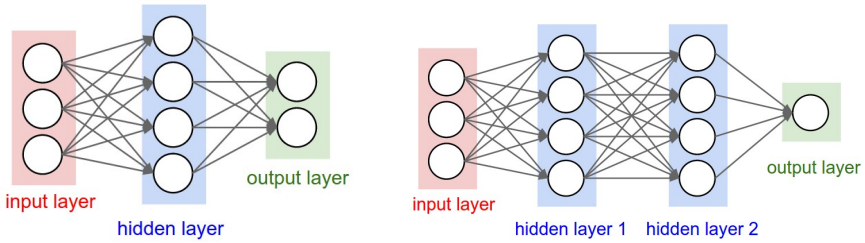
Machine Learning (ML) is considered a subfield of Artificial Intelligence (AI) that studies models capable of learning without being explicitly programmed to do so. Instead, the models rely on the learnt data and its patterns, so the end result is a model capable of making predictions. The adaptability of ML models makes them a great choice in scenarios where the data or the tasks are constantly changing, or when programming a conventional algorithm would be unfeasible. Inside of ML, there are three main approaches that depend on the nature of the supplied data: Supervised Learning (SL), Unsupervised Learning (UL) and Reinforcement Learning (RL).

- SL uses a labelled dataset to teach models to yield the desired output. This labelled dataset includes the inputs and the correct outputs, also known as ground truth. This dataset is provided to the model aiming that it learns the function that best maps the inputs and outputs, in order to make correct predictions on new and unseen data. This type of learning is mainly used for regression and classification problems.
- UL uses an unlabelled dataset. In this type of learning, the models have to find hidden patterns and trends in the data. This approach is mainly used for association and clustering problems.
- RL does not use a dataset. In RL, the model acts as an agent that learns by interacting with its environment. The agent receives rewards by performing correctly and penalties for performing incorrectly. The agent learns without intervention from a human by maximizing its reward and minimizing its penalty. Therefore, the models are built from the experience obtained by trying to solve the problem.

3.1.2 Artificial Neural Networks

Artificial Neural Networks (ANNs), first introduced by McCulloch and Pitts [12], are a mathematical model inspired by the biological neural networks and the way they process information. They are generally used to map non-linear input and output relationships, being extremely well-suited to solve problems in which the data is complex and noisy.

An ANN is organized into multiple layers of neurons that are interconnected with each other, analogously to the synaptic connections of the brain. These layers are usually divided into input layer, hidden layers and output layers. In cases where the network is composed of more than one hidden layer, they are generally labeled as Deep Neural Networks (DNNs).



(a) Artificial Neural Network (b) Deep Neural Network

Figure 3.1: Typical neural network architectures and difference between ANN and DNN convention [13].

3.1.2.1 Artificial Neuron

The core processing units of the ANNs are the neurons, also known as nodes, units or perceptrons (Figure 3.2). It receives as inputs the outputs from other neurons and computes an output itself. This output results in the application of one activation function in the weighted sum of the inputs and a bias (Equation 3.1).

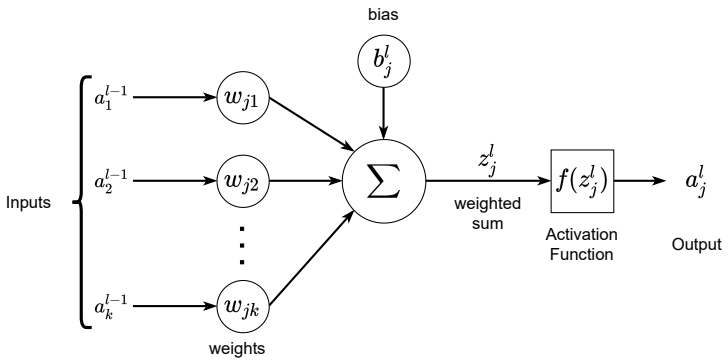


Figure 3.2: Artificial Neuron.

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$
$$a_j^l = f(z_j^l) \tag{3.1}$$

For each neuron, the following notation is used:

- w_{jk}^l : weight for neuron j in layer l for incoming node k
- b_j^l : bias for neuron j in layer l
- z_j^l : the weighted sum of input for node j in layer l
- a_j^l : output for node j in layer l
- f : activation function

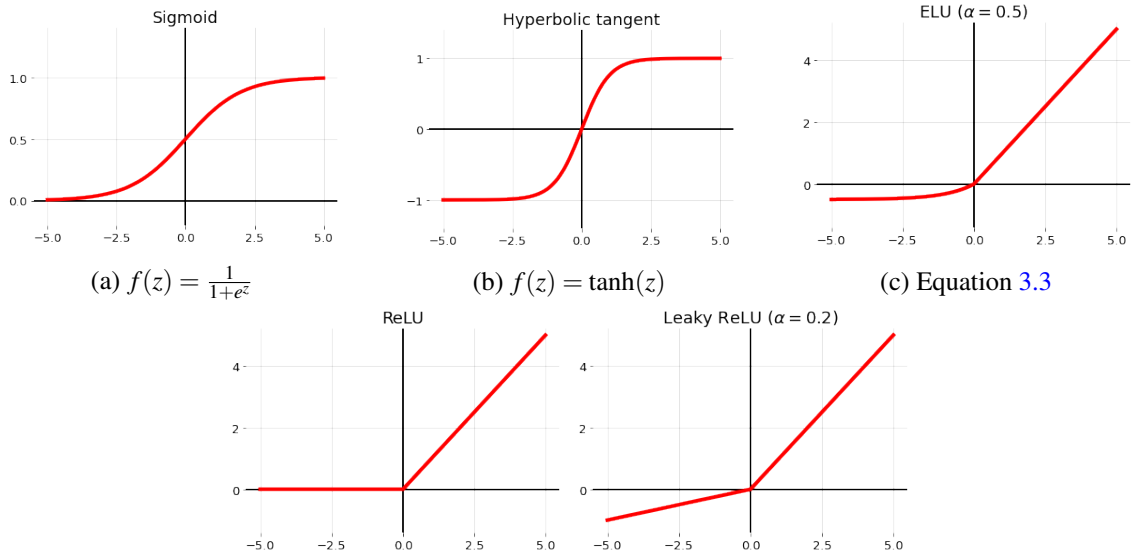
With this notation, the equation for the forward propagation can be rewritten in the vectorized form for each layer:

$$a^l = f(w^l a^{l-1} + b^l) \tag{3.2}$$

3.1.2.2 Activation Functions

The activation function determines the output of the neuron respectively to its inputs, deciding if the neuron should be activated or not. One of the reasons why non-linear functions are preferable over linear is due to the fact that non-linear functions have a non-zero derivative that makes the backpropagation algorithm [14] computationally efficient. Moreover, the addition of nonlinearities allows the network to learn complex data with high levels of accuracy. Also, they are used to normalize the output for a given range.

Figure 3.3 shows some of the commonly used functions and their respective equations. First, it can be seen that sigmoid normalizes the output between 0 and 1, whereas the hyperbolic tangent function is centered at zero and normalizes between -1 and 1. However, both of these functions are subject to the problem of vanishing gradient, i.e., when the network refuses to learn further, becoming too slow to make an accurate prediction. As such, Rectified Linear Unit (ReLU) [15] and one of its variants, Leaky ReLU [16], emerged to fix this problem. The latter avoids the dying ReLUs problem by providing a slope (α) in the negative part of the function. In addition, Clevert et al. [17] introduced the Exponential Linear Unit (ELU) which speeds up the learning process in deep neural networks and leads to higher classification accuracies compared to the ReLU and its variants. Last but not least, there is the softmax activation function, typically used in the output layer in classification problems. This function normalizes the outputs for each class between 0 and 1, and divides them by their sum, giving the probabilities of each input value belonging to a specific class.



(a) $f(z) = \frac{1}{1+e^z}$ (b) $f(z) = \tanh(z)$ (c) Equation 3.3

(d) $f(z) = \max(0, z)$ (e) $f(z) = \max(\alpha z, z)$

Figure 3.3: Typically used activation functions.

$$f(z) = \begin{cases} \alpha(e^z - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases} \tag{3.3}$$

3.1.2.3 Loss Functions

In ML, the loss function (also known as cost or error function) are used to evaluate how poorly models are performing. Simply, it is a measurement of how wrong the model is in terms of its ability to estimate the relationship between the inputs and outputs. This is typically expressed as a difference or distance between the predicted value by the ANN and the ground truth. Given a dataset $\{D = (x_n, y_n), n = 1, ..., N\}$, the lost function can be estimated by iteratively running the model to compare the estimated predictions $\hat{y}$ with the ground truth y, when the network is fed with the input vector x.

The two most commonly used functions are the Mean Squared Error (MSE) (Equation 3.4) and the Cross-Entropy (Equation 3.5) for regression and classification problems, respectively.

$$C(\theta) = \frac{1}{N} \sum_{n=1}^{N} (\hat{y}(x_n) - y_n)^2 \tag{3.4}$$

$$C(\theta) = -\frac{1}{N} \sum_{n=1}^{N} y_n \cdot \log(\hat{y}(x_n)) + (1 - y_n) \cdot \log(1 - \hat{y}(x_n)) \tag{3.5}$$

These functions are used not only for performance evaluation of ML models, but also in the training process to find the set of weights w^l_{jk} and biases b^l_j (collectively denoted as θ[1]) that minimizes the prediction's error.

[1] θ is often used in the literature as a notation for the ANN parameters

3.1.2.4 Training

Given a training dataset D, an ANN learns how to correctly map inputs to outputs through the training process. This process involves adjusting the weights and biases of the network that minimize the loss function. This is achieved by the backpropagation algorithm, that calculates the gradient of the lost function respectively to each parameter of the network. Then, the calculation proceeds backwards through the network propagating the error to the previous layers. This enables every parameter of the network to be updated individually, leading to a gradual reduction in the loss function at each training epoch. This process performs with the aid of one iterative optimization algorithm, as is the gradient descent.

In this section it will be reviewed some important concepts regarding ANN training, more precisely the backpropagation and gradient descent algorithms and the regularization methods to avoid the so-called overfitting.

Backpropagation

The most frequently method to train an ANN is the backpropagation algorithm [14]. It is usually used along with the gradient descent optimization algorithm, which is one the most popular and used. There are three variants of gradient descendent depending on the number of training samples [18]:

1. **Batch Gradient Descent (BGD)**

   This approach is the standard gradient descent. It computes the gradient of the loss function with respect to the parameters θ:

   $$\theta = \theta - \eta \cdot \nabla_\theta C(\theta) \tag{3.6}$$

   Where η is the learning rate, used to control the learning speed.

   In this algorithm the gradient is computed to the entire dataset before performing one update. This can be rather slow for bigger datasets or even unfeasible to perform when the datasets do no fit in memory.

2. **Stochastic Gradient Descent (SGD)**

   With stochastic gradient descent, rather than passing through the entire training set for a single update like in BGD, it performs a parameter update for each training sample $x^{(i)}$ and respective label $y^{(i)}$.

   $$\theta = \theta - \eta \cdot \nabla_\theta C(\theta; x^{(i)}; y^{(i)}) \tag{3.7}$$

This method solves the problem when the dataset is too large to fit in memory and, apart from that, it has faster convergence, as this prevents similar values from being computed on each parameter update. Nevertheless, it has some fluctuations in the minimization of the loss function. This allows it to reach new and possibly better local minima, though it can hinder the convergence to a global minimum, as it will keep overshooting.

3. **Mini-Batch Gradient Descent (MBGD)**

   In between, there is MBGD that takes the best of the other two variants. Instead of updating the parameters for every training sample, the updates are performed for every mini-batch of n training examples:

$$\theta = \theta - \eta \cdot \nabla_\theta C(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{3.8}$$

   With this approach the variance of the parameters update are reduced, leading to a more stable convergence. Also, by using batches, the gradient calculation is computationally more efficient than in SGD. This is the most popular gradient descent variant used in deep learning with the term SGD being often employed when batches are used.

As it can be seen, all the respective equations for each gradient descent variant have the variable η multiplied with the gradient of the loss function. This hyperparameter, called learning rate, is one of the most important when it comes to the ANNs training [19], as it controls the step size by which the parameters of the network are updated at each iteration. The parameters vector is moved in the negative direction of the gradient in order to reduce the lost function. There is always a trade off in the choice of this hyperparameter because low learning rates usually require more training epochs, due to the smaller changes made to the weights at each update, whereas high learning rates lead to rapid changes and require fewer training epochs. Moreover, lower learning rates also might never converge or may hold the process in an undesired local minimum, while higher learning rates can lead to fluctuations around a minimum or even divergence. Therefore, tuning the learning rate is a difficult process and has a great influence on accelerating the learning process. However in the recent years some approaches that dynamically change the learning rate during training have been developed to circumvent this problem. These approaches include learning rate schedules [20], cyclical learning rates [21], or adaptive learning rates, as is the case with the most common optimizers Adam [22], Adagrad [23], AdaDelta [24] and RMSProp [25].

In summary, it is through the optimization algorithms of gradient descent that the network parameters' values that minimize the loss function are found. However, it is through the backpropagation algorithm that the gradient of the loss function is propagated, from the output layer to the input layer, and the weights and biases of the network are adjusted.

Adapted from the book [26], the backpropagation algorithm using MBGD, in particular for a batch with m training examples, can be described by the following steps:

1. **Input a set of training examples**
2. **For each training example** x **:** Set the corresponding activation $a^{x,1}$ for the input layer and perform the following steps:
   (a) **Feedforward :** For each $l = 2,3,...,L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = f(z^l)$.
   (b) **Output Error** $\delta^{x,L}$**:** Compute the vector $\delta^{x,L} = \nabla_a C_x \odot f'(z^{x,L})$.
   (c) **Backpropagate the error:** For each $l = L-1, L-2, ..., 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot f'(z^L)$.
   (d) **Gradient:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^{x,l}_{jk}} = a^{x,l-1}_k \delta^{x,l}_j$ and $\frac{\partial C}{\partial b^{x,l}_j} = \delta^{x,l}_j$.
3. **Gradient descent:** For each $l = L, L-1, ..., 2$ update the weights according to the rule $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$, and the biases according to the rule $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$.

Regularization

One of the main problems that occur during the training process is overfitting. This phenomenon is described when the model has good performance on the training data but poor performance on new unseen data, meaning that the model failed to generalize. Frequently, the more complex the model, for example, by adding more intermediate layers or neurons, the more prone it will be to overfitting. On the contrary, an increase in training samples can be beneficial to reduce overfitting which can be done through Data Augmentation. This technique involves expanding the existing dataset by performing transformations, such as flipping, cropping or rotating on the training instances for artificially creating more samples. Other examples of regularization methods often used in deep learning to avoid overfitting are:

- **Dropout:** This regularization technique was introduced by Srivastava et al. [27] and is illustrated in Figure 3.4. By using Dropout, some of the neurons are "turned off" or omitted randomly with a probability p (dropout rate) at each training step. This prevents the neurons from co-adapting too much on the neighbours, forcing the network to learn different paths to each output. This way it allows the model to generalize better. At the end of the training process, all the nodes become active again.

(a) Standard Neural Network (b) After applying dropout

Figure 3.4: Example of dropout application. Left: A standard neural network with 2 hidden layers. Right: An example of dropout application on the network on the left [27].

- **Early stopping**: With this technique, the performance of the model is frequently monitored on the validation set (e.g. at every 5 training epochs) and the training is stopped as soon as the performance starts to decrease. This avoids the network from training too much to the point of starting to overfit the training dataset.

- **Batch normalization:** This technique was first introduced by Ioffe and Szegedy [28] with the goal of reducing the internal covariate shift [2]. This not only affects the learning process by highly reducing the number of training epochs required to train deep neural networks, but also provides regularization, avoiding overfitting. Besides, it also allows the use of higher learning rates without the exploding or vanishing gradients problem. With this regularization method, the mean and the variances of the layer's input are adaptively normalized at each mini-batch in the training process. The algorithm of this regularization technique is described in Algorithm 1.

Algorithm 1 Batch Normalizing Transform, applied to activation x over a mini-batch [28]

Input: Values of x over a mini-batch: $B = \{x_{1\ldots m}\}$; Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ // mini-batch mean
$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$ // mini-batch variance
$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$ // normalize
$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$ // scale and shift

3.1.3 Convolutional Neural Networks

The Convolutional Neural Network (CNN), introduced by Lecun et al. [29], is a deep learning architecture popularly used for image classification and object detection. Their ability to recognize patterns and features from images eliminates the need for manual feature extraction by traditional computer vision algorithms, making them very powerful and useful for several image and video applications, such as medical image analysis or self driving cars [30]. Although, they can also be effective for classifying non-image data such as audio [31] or time series.

A CNN can be split into two parts: The first is the feature extractor part composed of convolutional and pooling layers, and the second being the classification part composed of fully connected layers as a typical ANN. The Figure 3.5 depicts a typical CNN architecture with 2 convolutional and 2 pooling layers, followed by a fully connected layer and typically ending with a softmax layer as the output layer, whether it is a classification problem.

[2] The authors [28] refer to the internal covariate shift as the change in the distribution of network activations due to the change of its parameters during the training process. When the input distribution changes, hidden layers try to learn to adapt to the new distribution, slowing down the training process.

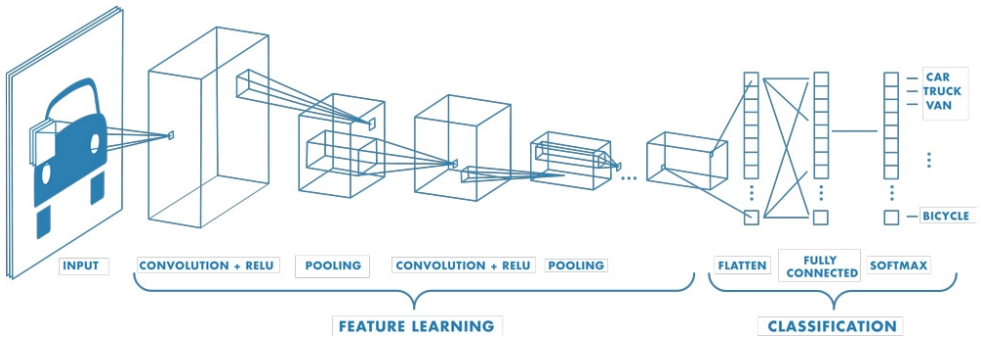


Figure 3.5: Typical CNN architecture. [32].

3.1.3.1 Layers

1. **Convolutional Layers**

   The convolutional layer is the main building block of a CNN and is responsible for the feature extraction. This is achieved through the convolution operation which is the main operation of these types of layers. Usually, a CNN is composed of several of these layers where the early ones extract low-level features and pass them to the later ones that extract high-level ones. Plus, these layers are generally followed by activation layers (typically ReLU) and pooling layers.

   The core units of these layers are the kernels (or filters) which are matrices with learnable parameters just like the weights in a typical ANN. These kernels perform the convolution operation by sliding over the input image performing the dot product at every patch of the image and storing the results in an output matrix called feature map or activation map. Figure 3.6 illustrates the convolution operation on a 2D input matrix.

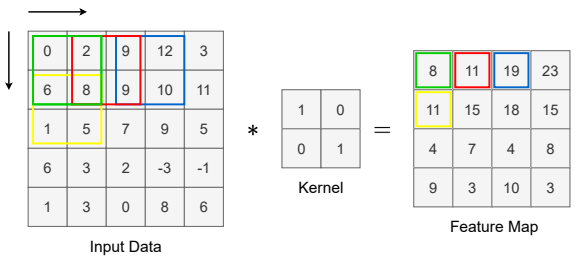


Figure 3.6: Convolution between a 5x5 input image and a 2x2 kernel with stride 1. The result is a 4x4 feature map.

Therefore, a convolutional layer generally takes as input an image of dimensions $I \times I \times C$, where I is the height and width of the image and C is the number of channels (for example, in an RGB image the number of channels is 3). Furthermore, the convolutional layer is composed of K kernels of dimensions $F \times F$, where F is smaller than the dimensions of the input image ($F < I$). Each of these kernels has different weights and is used to detect

different characteristics in the image, i.e., one of the kernels may be used to detect corners whereas other may be used to detect edges. The K features maps resulting from the convolution of the kernels with the input image are aggregated and form the output volume of the convolutional layer, which is passed to the following layers. The depth of the output volume is equal to the number of filters K. At the same time, the height and width (O) of the feature maps depend not only on the size of the filter (F) mentioned above, but also on 2 other hyperparameters: Stride (S) and Padding (P). On one hand, the Stride indicates the number of pixels to the right and down by which the kernel moves after each operation. On the other hand, Padding refers to the number of pixel borders added to the input image when it is being processed by the kernel. The most common padding operation is usually the zero padding. The zero padding works by extending the area of the image, adding zero value pixel borders and giving more space for the kernel to cover the image and thus, allowing a more accurate extraction of the features. Finally, the height and width (O) of the feature map are given by the following equation:

$$O = \frac{I - F + 2P}{S} + 1 \tag{3.9}$$

2. **Pooling Layers**

   These layers are usually present after a chain of convolutional layers and are responsible for reducing spatial dimensions, thus reducing the amount of computational power needed. They are useful for extracting dominant features and causing less overfitting and training time. The most used operations are the max pooling and the average pooling, which take the maximum value and the average value of each window of the feature map. Figure 3.7 illustrates both of these pooling operations.

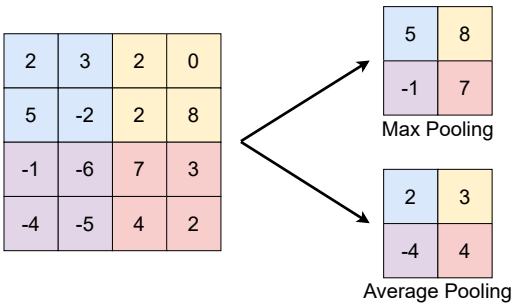


Figure 3.7: Max and Average pooling operations with 2x2 filter and stride 2.

3. **Fully Connected Layers**

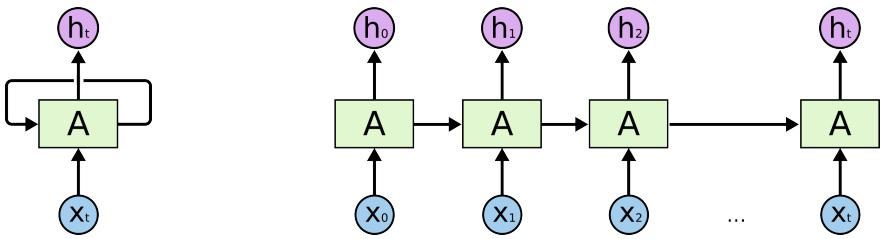
   This layer behaves similarly to a typical ANN described earlier in Section 3.1.2 . These layers are usually found at the end of a CNN architecture, taking as input the results of the previous convolutional and pooling layers and assigning a probability of belonging to a particular label or a single value, whether it is a classification or regression problem,

respectively. To this end, the output volumes are flattened into a single vector of values and then fed into the fully connected layer. After passing through this layer, the final layer uses the softmax function to get probabilities of the input being in a particular class, or another function (e.g. TanH, sigmoid or linear) to output a crisp value.

3.1.4 Recurrent Neural Networks

The Recurrent Neural Networks (RNNs) are a class of neural networks suited for processing time-series data or sequential data. They differ from feedforward neural networks, by having feedback connections to themselves (loops) which allow previous outputs to be used as inputs, while having hidden states. So, in this case, the inputs are related to each other and not independent and their internal memory capacity makes them very useful to solve temporal problems, such as speech recognition [33] and language translation.

Figure 3.8a illustrates the loop representation of an RNN cell where in a certain time step t, takes some input x_t and an internal state from a previous time step h_{t-1} and then it computes an output h_t. In addition, this output is used as an internal state update passed to the cell in the next time step. In a similar way, by unrolling the loop through time, the RNN can be seen as a direct cyclic graph with multiple cell copies passing a message to a descendent from one timestep to the next (Figure 3.8b).



(a) RNN (b) RNN unrolled through time

Figure 3.8: An RNN loop representation and its unrolled through time equivalent. Adapted from: [34].

The training process of RNNs usually uses a variant of backpropagation, called backpropagation through time [35]. The algorithm is similar to the backpropagation mentioned earlier in Section 3.1.2.4, but here it takes into account the unrolling of the network through time. In this case, errors are calculated and accumulated for each timestep and then they are backpropagated from the last time step to the first one. This means that an increase in the number of timesteps tends to be computationally and memory expensive. Moreover, this can expose the gradient vanishing problems which makes the memory of these networks retain only short-term dependencies. This results from the low-value gradients preventing the error to spread over time. As a consequence, the network does not learn further and forgets the long-term-dependencies between present and past information.

3.1.5 Long Short Term Memory

The Long Short Term Memory (LSTM) is an RNN architecture first introduced by Hochreiter and Schmidhuber [36] in 1997. Currently, is the most effective and used architecture to process sequences of data or time-series and were designed to overcome the vanishing and exploding gradient problems of the RNNs.

The architecture of an LSTM unit is depicted in the Figure 3.9. The main concept of these units is the cell state that works as a memory element. The flow of information in the unit is regulated by 3 gates:

- **Forget gate:** it decides what information from prior states should be kept in the cell state. It takes the prior hidden state h_{t-1} and the current input x_t and applies the sigmoid function. The value f_t is then between 0 and 1, where a value closer to 1 means to retain while closer to 0 means to forget.

- **Input gate:** it decides which values from the current step are relevant to be stored into the cell state. Same as the forget gate, it takes h_{t-1} and x_t and applies the sigmoid function, but this time 0 means irrelevant while 1 means relevant. Then, h_{t-1} and x_t are also passed through a TanH function creating a vector of candidate values $\tilde{C}_t$ used to help regulate the network.

- **Cell state:** based on the information from both forget gate and input gate, it selectively updates the values into the new cell state C_t. First, the prior cell state C_{t-1} is multiplied by the forget vector. This decides whether the previous values get dropped in the cell state when multiplied by values closer to 0. Then the output from the input gate is taken and the addition operation updates the cell state to new relevant values C_t.

- **Output gate:** The output gate determines the value of the next hidden state h_t. First, the h_{t-1} and x_t are once again passed through a sigmoid function giving o_t. Then, the new cell state C_t is passed through a TanH function. The results of both operations are multiplied and compose the information that should be carried by the hidden state h_t.

This description of the process in each of the gates and in the cell state is supported by the equations for a forward pass in an LSTM unit:

$$\begin{aligned} f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\ i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\ \tilde{C}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \\ C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\ h_t &= o_t \odot \tanh(\tilde{C}_t) \end{aligned} \tag{3.10}$$

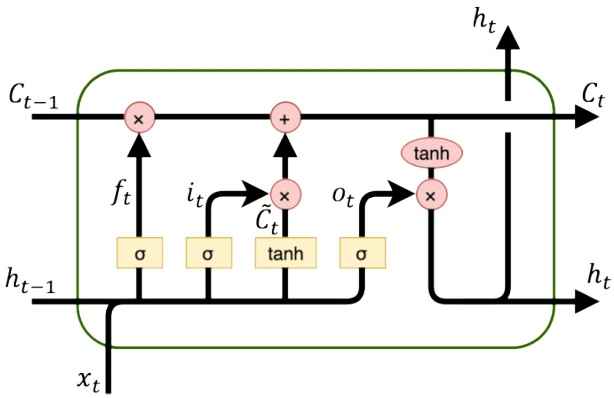


Figure 3.9: LSTM unit. The rectangles represent the neural network layers with their respective activation functions, while the circles denote pointwise operations ($\odot$). The vectors are represented by the lines, with vector concatenation and copying corresponding to line merging and forking, respectively [37].

In these equations, $\odot$ represents the Hadamard product, x_t the input vector and h_t the hidden state vector or output vector. Besides, W_x represents the matrix of weights associated with the input pairs x_t, U_x the matrix of weights associated with the prior hidden state h_{t-1} and b_x the bias vector, where x represents the respective gate between the forget f_t, input i_t and output o_t gates and the candidate hidden state $\tilde{C}_t$. These weights and biases need to be learned during training.

The Figure 3.9 and the Equations 3.10 represent the most frequently implemented LSTM variant that uses a forget gate [38]. However, some other variants have emerged throughout the years. Namely, the introduction of peephole connections [39] and, more recently, of the Gated Recurrent Units (GRUs) [40]. Although simpler than a typical LSTM, they have been quite popular, revealing better performance in smaller training datasets.

3.1.6 End-to-End Learning

End-to-end learning is the process in which a complex system is represented only by a single learning algorithm that learns all the intermediate steps from the initial phase of the input to the final phase of the output. Generally, in this learning mode, the complex system is replaced by a single DNN that is responsible for learning the entire control pipeline directly from the input sensory data. This is efficiently made by tuning the parameters of the entire model based on the correctional signal from the output. In this approach, all the intermediate steps of the system are trained simultaneously and not sequentially. In a short manner, an end-to-end system maps the raw input directly to the outputs, bypassing the intermediate stages usually present in a traditional pipeline.

However, some authors [41, 42] point out some drawbacks of this type of approach. Firstly, as in end-to-end learning prior knowledge is not integrated, they mention the need to provide a large and diverse training dataset to allow the acquisition of this knowledge. Secondly, they also reveal

that with an increase of system's complexity, learning in an end-to-end manner can become inefficient or even fail. As such, Shalev-Shwartz et al. [42] support a "Divide and conquer" approach considering that the training of complex learning machines should proceed in a structured manner, training simple modules first and independent of the rest of the network [41].

In any case, this approach has been used with great success in several fields of application. For example, speech recognition applications [33, 31], robotics [43] and self driving cars, as shown in Figure 3.10 and also as will be described later in Section 3.2 by showing some state-of-the-art architectures for this last purpose.

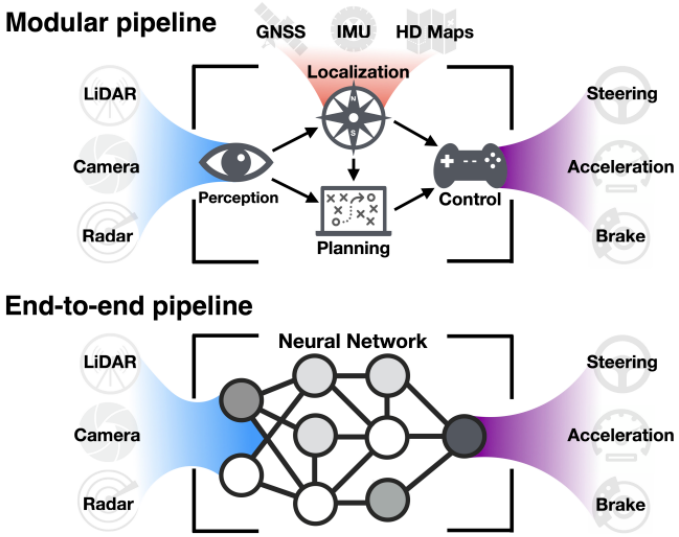


Figure 3.10: Traditional and end-to-end pipelines for self driving cars. The traditional pipeline for autonomous driving consists of many interconnected modules, while end-to-end approach treats the entire pipeline as one learnable machine learning task [44].

3.2 State of the Art

In this section, it will be presented the state of the art regarding deep learning and also some of its applications in the field of autonomous driving.

3.2.1 CNN architectures

As mentioned before in Section 3.1.3, a typical CNN architecture comprises a couple of convolutional layers followed by ReLU and pooling layers, ending with fully-connected (dense) layers and output layer.

The way these layers are organized within the network, as well as the size and number of kernels and the hyperparameters to select, are targets of great interest and research by the scientific community. This growth in interest led to the conception of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC)[3], an annual competition that promotes the development of object detection and image classification algorithms using a subset of the ImageNet database[4]. This database contains about 14 million images and about 21 thousand classes and, together with ILSVRC, led to the development of the most traditionally used CNN architectures such as AlexNet [45], VGG-16 [46], GoogeLeNet [47], and ResNet [48]. All these state-of-the-art architectures arose after LeNet-5 [29] by Yann LeCun in 1998, widely used for written digits recognition (MNIST[5]).

The research in [49] overviews the state-of-the-art deep learning architectures, including those mentioned in the last paragraph and points out the differences between them. These differences are summarized in Table 3.1.

Table 3.1: Comparison between the different state-of-the-art CNN architectures. The top-5 error stands for the metric classification taken in the ILSVRC on the ImageNet and MAC for Multiply-Accumulate operations. Adapted from [49].

Methods	LeNet-5	AlexNet	VGG-16	GoogleLeNet	ResNet50
Top-5 errors	n/a	16.4	7.4	6.7	5.3
Input size	28 x 28	227 x 227	224 x 224	224 x 224	224 x 224
Nº of Conv Layers	2	5	16	21	50
Filter Size	5	3,5,11	3	1,2,5,7	1,3,7
Nº of Feature Maps	1,6	3-256	3-512	3-1024	3-1024
Stride	1	1,4	1	1,2	1,2
Nº of Fully Connected layers	2	3	3	1	1
Total Weights	431 k	61 M	138 M	7 M	25.5M
Total MACs	2.3 M	724 M	15.5 G	1.43 G	3.9G

[3] http://www.image-net.org/challenges/LSVRC/. [Last accessed in 16/06/2021]
[4] http://www.image-net.org/. [Last accessed in 16/06/2021]
[5] http://yann.lecun.com/exdb/mnist/. [Last accessed in 16/06/2021]

3.2.2 PilotNet

The ALVINN project [50] was one of the the first attempts to use an ANN for autonomous navigation. It used a 3-layer network that took images from a camera and a laser range finder as inputs. The network produced the direction the vehicle should travel to follow the road as output. Although it was a simple approach, its success suggested the potential use of neural networks for autonomous navigation.

Inspired in this work done with ALVINN, Bojarski et al. [30] from Nvidia proposed a deeper CNN to map raw pixels from a single front-facing camera directly to steering commands. The motivation was to eliminate the need for hand-coding rules and instead create a system that learns by observing [51] and correctly accomplish the lane following task. The proposed network architecture, called PilotNet, is shown in Figure 3.11.

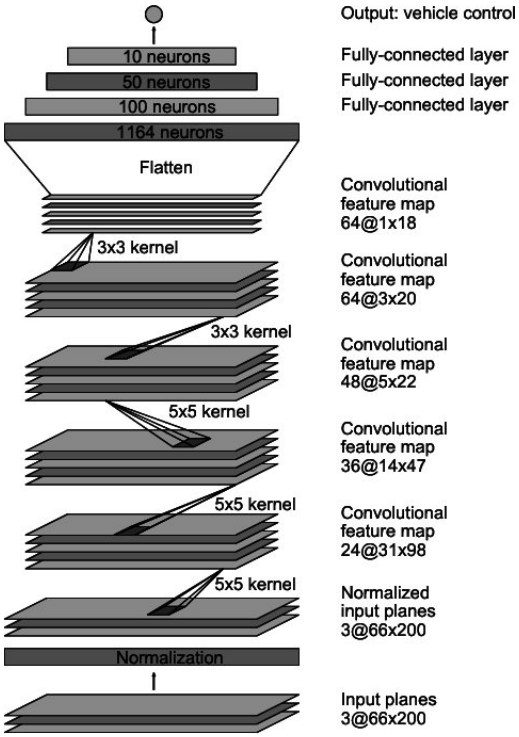


Figure 3.11: PilotNet architecture [30].

The network consists of 9 layers, including a normalization layer, 5 convolutional layers and 3 fully connected layers, which results in 27 million connections and 250 thousand parameters. This architecture is similar to AlexNet [45] which also uses 5 convolutional layers and 3 fully connected layers. However, AlexNet uses a much larger filter size in the first layer and makes heavy use of pooling layers.

Before being fed to the first convolutional layer, the 66x200x3 images in the YUV color format are adjusted to have a zero mean and unit standard deviation. This is done by the normalization layer, which has the function of performing image normalization and enable a more stable and

faster training process of the network [52]. It should be noted that this architecture does not use pooling layers as is common in CNNs. Instead, PilotNet performs feature map reduction along the network by a stride greater than one on the convolution layers. While these convolutional layers perform the feature extraction of the road, the following fully connected layers are responsible for output a steering angle. Both of these layers are adjusted during the learning process, except the normalization layer.

Respectively to the training process, the dataset was composed of pairs of steering commands and images collected from 3 cameras in driving scenarios with a wide variety of lighting, weather conditions and road types. Finally, both simulation and on road tests revealed very successful with a high level of autonomy, concluding that PilotNet perception is similar to human drivers.

Not only the project DeepPiCar from Bechtel et al. [53] but also the project ROTA from Rodrigues [54] are successful applications of this CNN architecture. The former is a small scale robot that proved to drive itself in real-time using a web camera and a Raspberry Pi 3. While the latter is a small robotic vehicle that participated in the PRO 2018 edition having won the 3rd place of the ADC. The trained network showed good performance both in a simulation environment and in the real track.

3.2.3 J-Net

Kocić et al. [55] proposed a CNN architecture for autonomous driving that was lighter and more suitable to run on embedded platforms. The motivation was to keep the same performance as other state-of-the-art architectures but with a smaller number of parameters, in order to reduce the computational load and consequently enable its implementation in cheaper and smaller hardware with less processing power.

The proposed architecture, called J-Net, is depicted in Figure 3.12. It has 3 convolutional layers, each followed by a pooling layer, 1 flattened layer and 2 fully connected layers. The network takes an RGB image as input, which is normalized and cropped to eliminate unnecessary information besides the road.

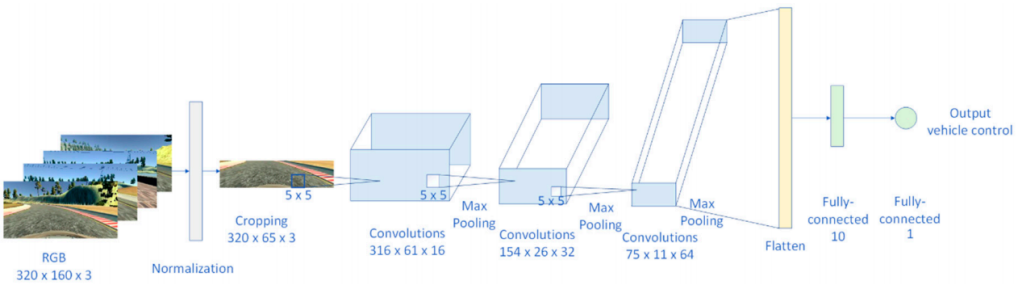


Figure 3.12: J-Net Architecture [55].

The network was compared with PilotNet [30, 51], and AlexNet [45] regarding the computational complexity and performance evaluation. Firstly, J-Net was 4 times smaller than PilotNet

and 250 smaller than AlexNet which led to an improvement in the real-time performance by comparison. Secondly, AlexNet had the best performance in general with smoother steering, better performance at the curves, and closer to the lane center most of the time. Although, this was justified by the fact of AlexNet having a deeper architecture than the other two. The 3 architectures successfully performed the task of autonomous driving, with J-Net having the smaller deviation from the center of the track on the curves.

3.2.4 Convolutional Long Short-Term Memory Network

Despite the positive results of the CNN based end-to-end approaches, including the PilotNet [30, 51], J-Net [55] and [56], in all of these algorithms temporal dependencies are not being considered. This means the steering angles are being predicted in each individual video frame, ignoring their temporal relationship.

A solution to circumvent this problem was proposed by Eraqi et al. [57], which consisted in a Convolutional Long Short-Term Memory Recurrent Neural Network (C-LSTM). This network is end-to-end trainable, to learn both visual and dynamic temporal dependencies of driving. The system architecture of the training process is shown in Figure 3.13.

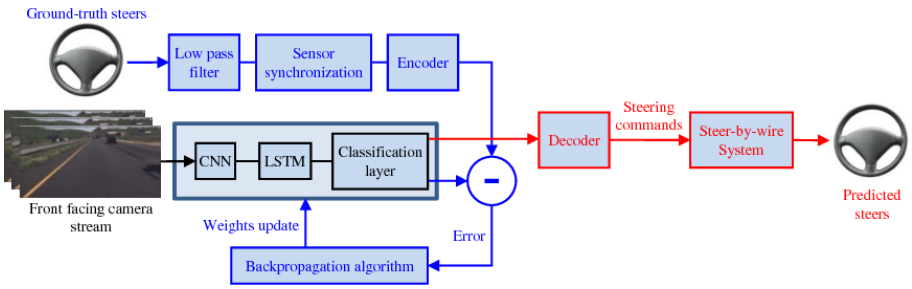


Figure 3.13: Training process of the C-LSTM model [57].

It is a compound neural network consisting of a CNN and an LSTM network that estimates the steering wheel angle based on the front-facing RGB camera as input. Camera images are processed frame by frame by the CNN. The resulting features are then processed within the LSTM network to learn temporal dependencies. The steering angle prediction is calculated by the classification layer after the LSTM layers. For full deployment, the steering angle prediction is transmitted to appropriate actuators by a steer-by-wire system. The author made a comparison between this model and the one proposed by Bojarski et al. [30] through training and validating both models with the publicly available dataset released by *comma.ai* [6][58]. The results revealed an improvement of the steering root mean square error by 35% and led to more stable steering by 87%.

The work of Fu et al. [59] revealed excellent results of this network architecture for steering a mobile robot. Although the input is a LIDAR and not an image, the authors concluded that this

[6] http://research.comma.ai/. [Last accessed in 16/06/2021]

method outperforms several existing deep learning methods by comparing their performance, revealing that this architecture is better at adapting to a new environment, better at imitation learning and lead to less overfitting.

Chapter 4

Proposed Approach for the Machine Learning Controller

In this chapter, a detailed description of the methodology for the navigation control system based on ML algorithms is presented. Firstly, the datasets preparation details are mentioned, secondly a description of PilotNet and JNet architectures is established, then, details about the training of these models, as well as the ones regarding the software and hardware utilized are described. Finally, the modifications made to the previous ROS architecture are described.

4.1 Overview

As seen in the Chapter 2, the Major Alvega robot driving control system is based on traditional computer vision, odometry and control theory methods, namely a PID controller. This system won the 2019 edition of the PRO's ADC held in Gondomar, Portugal. From this point forward this system will be referred to as Previously Developed System (PDS).

Despite the robot's good performance in the competition, the goal of this dissertation is to reimplement this control system with a more modern and advanced approache. Therefore, ML techniques, such as ANN and SL, will be used to control the robot steering.

Since the ANN will have, as input, an image captured by the tracking camera and also considering that it will have to predict a real number for the steering angle to be applied to the robot, the most suitable architecture for this regression problem is a CNN. Figure 4.1 depicts, in a simplified overview, the proposed controller that relies on a CNN that will have to learn how to control the robot in an end-to-end manner by using SL.

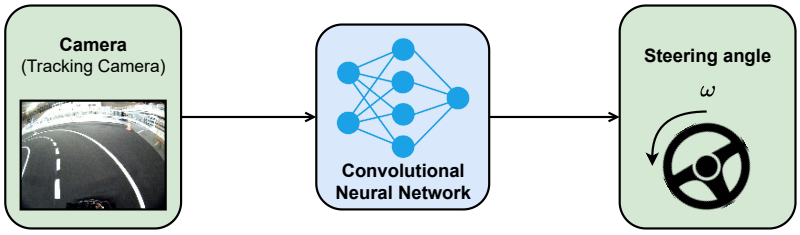


Figure 4.1: The proposed ML controller to the Major Alvega robot

To use SL, the training datasets must consist of a tracking camera frames (samples) and the respective steering angles that keep the robot inside the lane (labels). For this purpose, previous logs (ROS bags) of the robot driving in the real track were used to build these datasets (Section 4.2). In the situations where the available data was insufficient, the developed simulator was used to generate more data and thus enrich the dataset. To perform the D1, D2 and D3 challenges of the driving category, 4 distinct datasets were created to train 4 models: one running the track on the outer lane (Section 4.2.3), another in the inner lane (Section 4.2.4), and two in those lanes but in the opposite direction (Section 4.2.5). The raw steering angles went through a filtering process (Section 4.2.1) in order to remove the noise. To filter the relevant information to be fed to the networks, the images also underwent trough a pre-processing step (4.2.2). Furthermore, two different CNN architectures were tested: the PilotNet (Section 4.3.1) and the JNet (Section 4.3.2). These networks were trained on the datasets created (Section 4.4) and implemented as described in Section 4.5. Finally, the trained models were integrated into the ROS architecture of the PDS as explained in Section 4.6.

4.2 Dataset Preparation

As mentioned before, ROS bags were used to build the dataset. These bags included both relevant and not relevant situations for the network to learn how to drive properly along the track. These situations included crashes, idle moments, lane deviations, driving in reverse and obstacle deviation performance. To select only the data where the robot drives correctly, a visualization tool called webviz[1] was used. This web-based application allows the visualization and playback of ROS bag files. One of its main advantages is having a GUI that allows the user to select and organize the topics recorded in the bag file. This enables the user to get immediate visual insight into the recorded data. This monitoring includes plotting the content of the topics, visualizing the camera images, 3D point clouds, among others.

Thus, webviz and the command line tool `rosbag filter` were used to filter the relevant data, removing the irrelevant scenarios for the training data mentioned above. Figure 4.2 demonstrates the webviz GUI and its layout for viewing and selecting the relevant data to compose the dataset.

[1] https://webviz.io/app/. [Last accessed in 25/05/2021]

In this work, the images were recorded at 30 frames per second and with the robot driving at a constant velocity. These images were saved as JPEG in separate folders for each bag file, including a CSV file associating the image-steering angle pairs. This association was made through interpolation between the timestamps of the camera frames and the steering commands. The dataset is composed of a directory containing all these folders per bag and includes its own CSV file that concatenates all the information regarding the bags inside of it. This way, the association and representation of the whole dataset becomes simpler and easier to load.

Figure 4.2: Webviz GUI used for ROS bags visualizing.

When designing a neural network, one of the most relevant concerns that must be answered is what form both the input and output should take. Therefore, before performing the models' training process, the steering angle outputs underwent a filtering process (Section 4.2.1), and the input images passed through a pre-processing step (Section 4.2.2).

4.2.1 Steering Angles Filtering

The steering angles are the output of the network and the labels in the training process. In this work, the steering angles are represented in radians where a negative value represents a left turn while a positive value represents a right turn.

In the ROS bags, these angle measurements contained some noise due to some glitches or state transitions. Training the network directly with this noisy ground truth would be prejudicial to the learning process of the network. This is due to the fact that training a network with noisy labels tends to memorize information about the noise, which degrades the generalization performance [60].

To face these discrepancies and to improve the network's learning, the steering angles of the original system were filtered by a Low Pass Filter (LPF) in order to smoothen the raw steering angles signal. In this way, the network learns to infer a steering angle that keeps the robot within the lane instead of learning about the noise included in the signal. Without this noise, it is also expected that the robot learn to steer more smoothly than the PDS. Figure 4.3 shows the effect of

the LPF on the ground truth. The blue line represents the steering angles recorded in an example ROS bag where Major Alvega drove two laps on the outer lane in the real track. The orange line denotes the smoothed steering angles after applying the LPF, which constitute the new ground truth of the datasets.

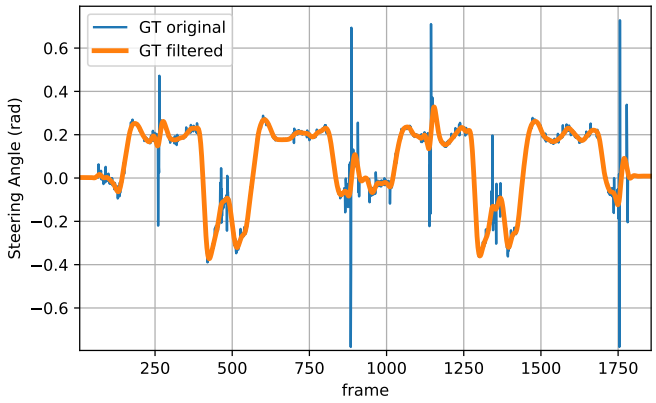


Figure 4.3: LPF effect on the ground truth.

In the situations where the robot travels on the inner lane, the original system performed the initial and final turns using open loop control. This would generate a high frequency step signal that would be completely filtered out by the LPF. To overcome this problem, the bags including the robot data traversing the inner lane were divided into two distinct situations: situations where the robot drives using closed-loop control and situations where the robot drives using open-loop control. The Figure 4.4 illustrates the zones where the system runs in open-loop and in closed-loop.

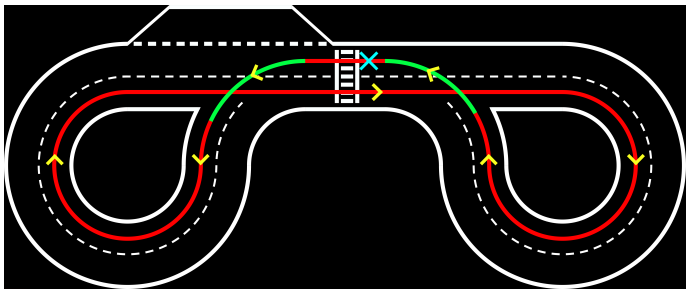


Figure 4.4: The green region has the robot driving using open loop control, whilst the red region has the robot driving using closed loop control. The blue X depicts the starting point.

In situations where the robot drives in closed-loop the LPF was applied as shown in Figure 4.3. In situations where the robot drives in open-loop, a median filter was applied. This way, the essential steering angles values in the turning maneuvers are kept, but the noise peaks are removed. Figure 4.5 demonstrates the application of the median filter on an example bag where Major Alvega was performing a final left turn on the second half of the track.

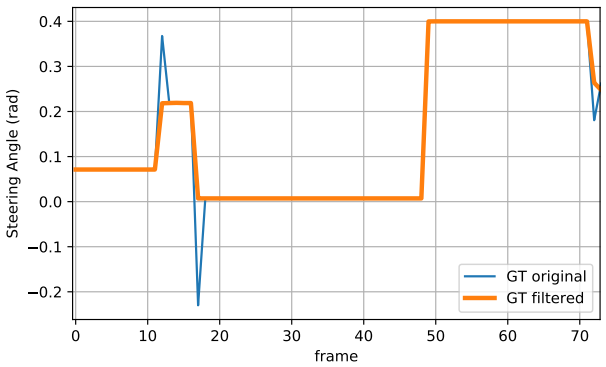


Figure 4.5: Median filter effect on the ground truth.

4.2.2 Images Pre-processing

When designing neural networks one must take into account the shape of the input vector. Furthermore, for an effective learning process it is important that the neural network is able to learn relevant features and also be capable of disregard the noise or unnecessary information that could be present.

The image captured by the tracking camera has a resolution of 640 x 480 pixels in the RGB color space. However, the image is resized by a scaling factor of 1/4 to decrease the number of pixels. Hence, the image used for processing has a final resolution of 160 x 120 pixels. The camera uses a fisheye lens, which has the advantage of increasing the field of view in such a way that both lanes can be observed simultaneously. However, it has the disadvantage of adding distortion to the image. Figure 4.6 shows an example of an image taken by the camera on the real track and in the simulation environment.

(a) In the real track

(b) In the simulation environment

Figure 4.6: Images captured by the tracking camera.

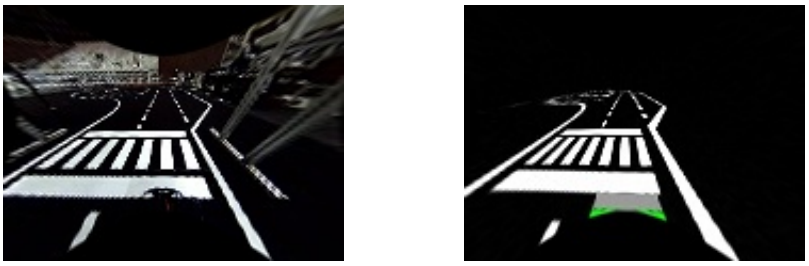
Before performing the training process or inference, the image captured by the camera goes through a pre-processing step according to the following transformations:

1. **Format conversion**

The image is received in the ROS Image[2] message format. To be processed, it is required to be converted to the image format used by the OpenCV library, using the cv_bridge[3] module.

2. **Fisheye distortion removal**

   In this step, the distortion created by the fisheye lens is removed as depicted in Figure 4.7. In this manner one obtains images that are less circular and more rectangular and regular but with the disadvantage of reducing the field of view [61]. This is achieved by calibrating the camera and obtaining its intrinsic parameters.



(a) In the real track (b) In the simulation environment

Figure 4.7: Images without fisheye distortion.

3. **Cropping**

   In this step, the parts of the image that are not essential for the driving task are discarded, in particular the area visible above the road. The removal of these parts reduces the noise, streamlines training by focusing on the most relevant details, and highlights the relevant information content of the input pixels [61]. The track pixels, several meters ahead of the current position are also not relevant for the network's angle prediction at that instant. Taking this into account, the region of interest of the image that is cropped has a size of 65x160 and is depicted inside a red rectangle in Figure 4.8.

(a) In the real track (b) In the simulation environment

Figure 4.8: Region of interest that is cropped.

[2] https://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/Image.html. [Last accessed in 27/05/2021]

[3] http://wiki.ros.org/cv_bridge. [Last accessed in 27/05/2021]

4. **Grayscale conversion**

   The track is mostly made of black and white colors. Therefore, the information from the three RGB channels does not bring much information to the network. Due to this, the image is converted to grayscale thus reduced to just one channel.

5. **Thresholding**

   To use the CNN models in both real and simulated environments, the image is thresholded. Since colors in a real environment are not ideal, in contrast to what happens in a simulated environment, this operation aims to reduce the difference between both environments. Additionally, the complexity of the input is reduced since there are only two colors: white for the lines and black for the rest.

   Figure 4.9 shows the final result after the pre-processing step on four example images.

Figure 4.9: Four examples of images after the pre-processing step

In brief, the input image provided to the network is then a binary image of size 65x120x1, containing only the relevant features for the network to learn how to drive the robot by predicting the proper steering commands. These images and their corresponding steering angles are then used to build the different datasets.

4.2.3 Outer Lane Dataset

To train the model to drive on the Outer Lane (OL) of the track, only the image-angle pairs traveling along this lane were used. This leads to the robot going straight ahead in the first half of the track and turning right in the second half, as shown in Figure 4.10. Since in this case the amount of data recorded on the real track was quite satisfactory, there was no need to record more data using the simulator.

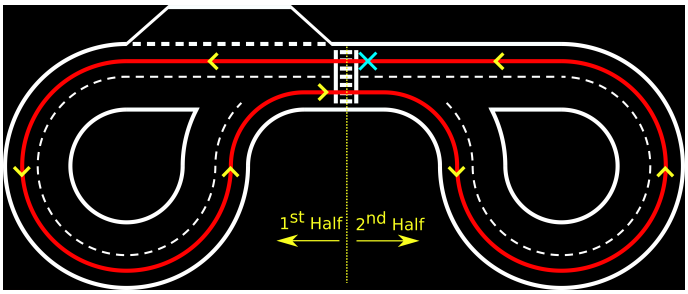


Figure 4.10: Outer Lane path used for the OL dataset. The blue X depicts the starting point.

The ROS bags used for this dataset were recorded with the robot driving at three different velocities. Table 4.1 shows the number of samples for each velocity, while Figure 4.11 represents the histogram of the steering angles' distribution used in this dataset.

Table 4.1: Number of samples per velocity on the OL dataset.

Velocities (m/s)	0.7	0.85	1.2	**Total**
Number of samples	1771	8656	10528	20955
Percentage (%)	8.5	41.3	50.1	100

In summary, the OL Dataset comprised approximately 12 minutes of driving data which resulted in a total of 20955 samples.

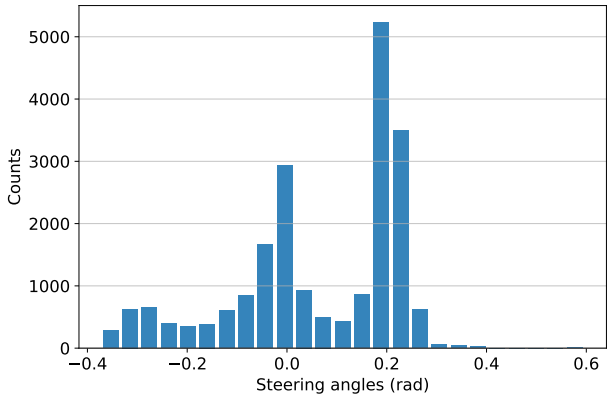


Figure 4.11: Histogram of steering angles distribution in the OL dataset.

4.2.4 Inner Lane Dataset

According to the challenge D2, the robot has to complete two laps around the track following the indications on the signaling panel. For this purpose, in addition to the dataset created to train a model to cover the OL, a second dataset was created to train a new model to drive on the Inner Lane (IL). This path, depicted in Figure 4.12, differs from the first in that the robot will have to

make a left turn at the intersection in the first half of the track and drive the curve through the inside lane with smaller radius. Besides this, in the second half of the track, the robot must be able to move forward on the straight leg, drive again the curve trough the inside lane with smaller radius, and make a left turn that returns it to the initial starting position, which is present in the OL.

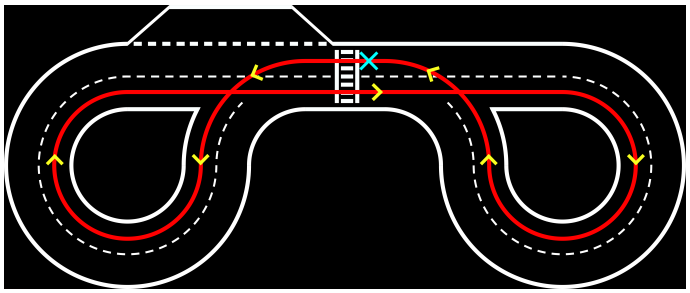


Figure 4.12: Inner Lane path used for IL Dataset. The blue X depicts the starting point.

For this path, the ROS bags recorded on the real track were not considered sufficient for training the robot to travel on the IL when compared to the OL Dataset. Therefore, the simulation environment and the PDS were used to create more data of the robot driving in this lane. The proportion of data recorded in the real track and in the simulation environment are given in Table 4.2.

Table 4.2: Number of samples in the IL Dataset recorded in the real track and in the simulation environment.

Track	Real	Simulated	**Total**
Number of samples	5534	12545	18079
Percentage (%)	30.6	69.4	100

Unlike the ROS bags used in the OL dataset, the ROS bags for this lane were all recorded with the robot driving at a velocity of 0.85 m/s, both on the real track and in the simulated environment. The dataset for the IL performance was composed of approximately 10 minutes of driving data resulting in a total of 18079 samples. Figure 4.13 represents the histogram of the distribution of steering angles used in this dataset.

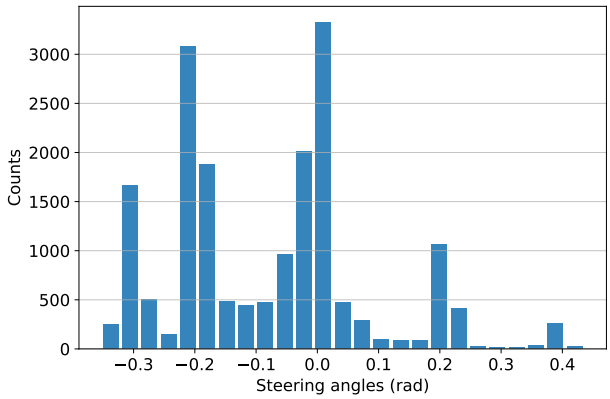
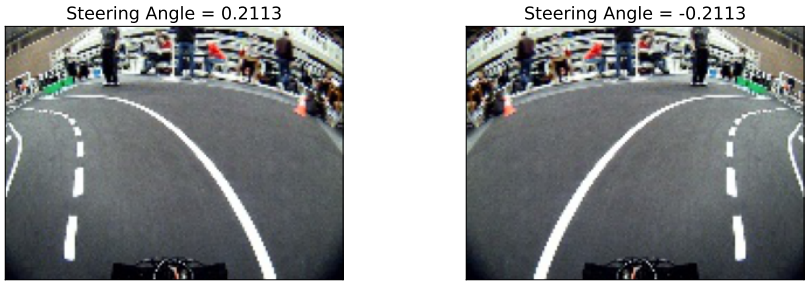


Figure 4.13: Histogram of steering angles distribution in the IL dataset.

4.2.5 Opposite Direction Dataset

According to the challenge D3, the robot must drive in the presence of obstacles in the second half of the track. Training a model able to circumvent obstacles would require colecting a dataset with the robot circumventing obstacles in different positions on the track. This solution would be time consuming and could probably require the use of a different neural network architecture dedicated to obstacle avoidance, as in [62].

Alternatively, it was taken advantage of the fact that the second half of the track is symmetrical, which allows to invert part of the data from the previous datasets, and enable the robot to drive in the opposite direction in both lanes. This inversion consists precisely in flipping horizontally the images and multiplying the steering angles by -1 as depicted in Figure 4.14. Therefore, this ultimately leads to the creation of two extra datasets: OL^{-1} dataset for driving the robot on the OL in opposite direction and IL^{-1} dataset for driving the robot on the IL also in the opposite direction.



(a) Original image (b) Horizontal flipped image

Figure 4.14: Horizontal flipping and symetric steering angle for driving in the opposite direction.

In this way, the architectures dedicated to driving were reused and to the system was given four different models trained with the respective datasets. In this manner, the four driving models are used to perform the obstacle challenge according to the simplified state machine shown in Figure 4.15.

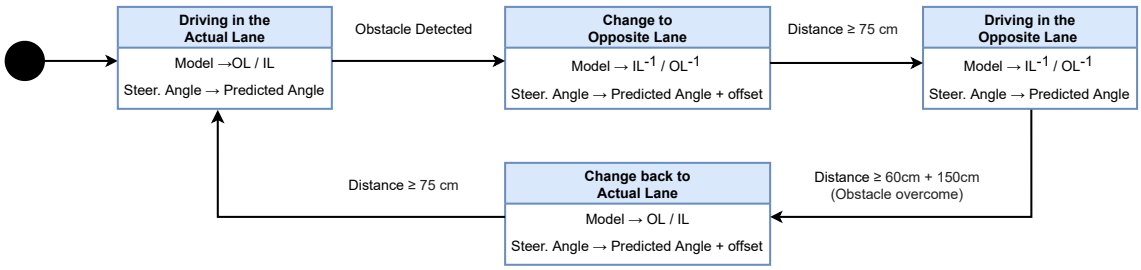


Figure 4.15: State machine employed for the obstacle avoindance.

When an obstacle is detected, a switch is performed between the model that drives in the current lane and the model that drives in the opposite lane. To perform the lane change maneuver, the speed is slowed down and an offset value is added to the steering angle value predicted by that model. This offset value forces the lane change, while the CNN output value gradually corrects the vehicle's orientation in the opposite lane. Based on the odometry data, the lane change maneuver is ended when a threshold distance of 75cm is reached. After that, the model driving in the opposite direction is maintained until a distance has been travelled which ensures with safety that the obstacle has been overcome. This distance is defined as a threshold value of 60 cm (length of the obstacle squared base) + 150 cm. Above this value, the driving model in the current lane is again reactivated. Similar to the first changing maneuver, the speed is reduced and an offset is added to the steering angle for the robot retake the initial lane and resume the planned trajectory until a new obstacle is detected or the crosswalk area is reached. These offset values were obtained empirically so that Major Alvega would be able to perform obstacle avoidance in most parts of the second half of the track.

Hence, from the previous OL and IL datasets, only the scenarios where the robot travels along the track in the area where the obstacles are placed were reused, as shown in Figure 4.16. By inverting both the images and the angles and training two different models for each of the lanes, it is expected that the robot will be able to traverse the track in the opposite direction.

Finally, the number of samples for each of the two datasets is represented in Table 4.3 while the distribution of the steering angles is presented in Figure 4.17.

Table 4.3: Number of samples for OL^{-1} and IL^{-1} datasets for driving in opposite direction.

Dataset	OL^{-1} Dataset	IL^{-1} Dataset
Number of samples	9195	9632

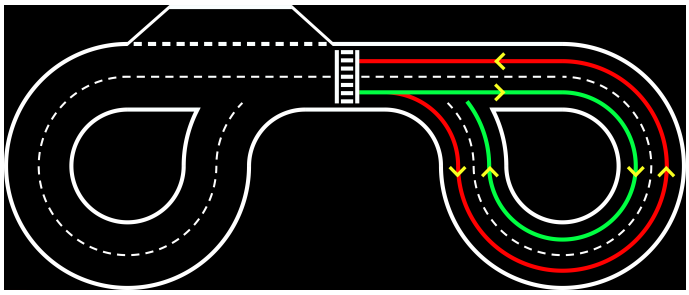
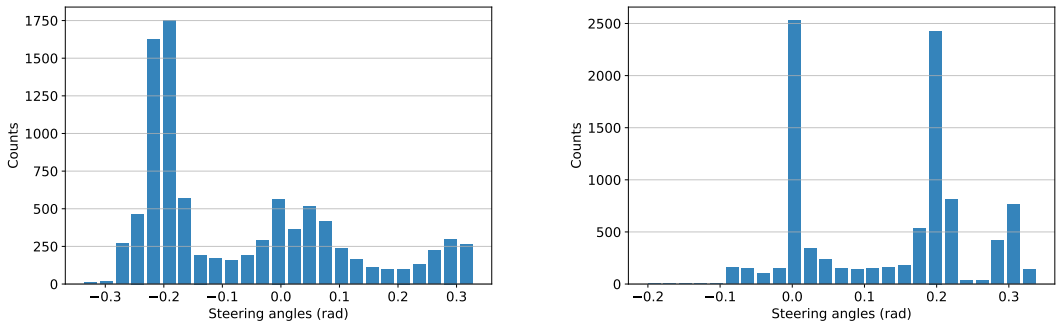


Figure 4.16: The red path represents the data from OL Dataset used to build OL^{-1} Dataset, in order to drive the OL in the opposite direction. The green path represents the data from IL Dataset used to build the IL^{-1} Dataset, in order to drive the IL in the opposite direction.



(a) OL^{-1} Dataset (b) IL^{-1} Dataset

Figure 4.17: Histograms of steering angles distribution in opposite direction dataset.

4.3 Model Architectures

Since images are being used to determine the robot's driving commands, the ANN architectures employed are CNNs. In order to test different architectures and to train various models for the driving performance of Major Alvega, two state-of-the-art architectures of CNNs used in autonomous driving were implemented: the PilotNet (Section 4.3.1) and the JNet (Section 4.3.2).

4.3.1 Pilot Net

One of the architectures implemented to infer the steering angle was based on the Pilot Net architecture. Whilst in the original paper [30] three cameras were used to train the network and only the central camera was used to infer the angle, in this project only the tracking camera was used for both situations.

To reduce the size of the feature maps there are usually two options. The first is to use a stride greater than 1 in the convolutional layers while the second is to use max pooling layers after the convolutional layers. In the original paper [30] the architecture uses a stride greater than 1 in the

first 3 convolutional layers. However, in this project max pooling layers were used with a 2x2 size filter after the first 3 convolutional layers. At an early stage, the first option was implemented, but better results were obtained using the max pooling layers option. Also the authors do not refer in the paper to any regularization method they used during the training. In this case, dropout layers were added after all fully connected layers with a dropout rate of 20% for the purpose of reducing the overfit. As with the original architecture, the normalization layer was kept as the first layer of the network, and in this case it is used to normalize the image between 0 and 1. This ensures the same range of values for each of the inputs, which leads to a stable convergence of weights and biases. This layer was employed using Lambda layer[4] in Keras.

Since the steering angle is a real value in radians that can be either negative or positive, in the output layer the linear function was used as the activation function.

The architecture of the network employed is illustrated in Figure 4.18, and the specification of the layers is shown in Table 4.4. The network has a total of 545419 trainable parameters.

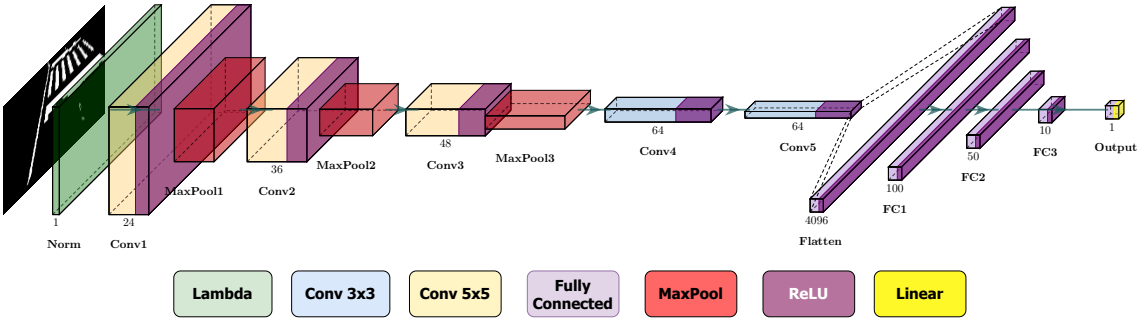


Figure 4.18: Visual representation of the PilotNet architecture.

4.3.2 J Net

The J Net architecture [55] was also tested for driving the Major Alvega robot. This network is different from the former by being less deep. This is due to the fact that this architecture is composed of only 3 convolutional layers, 3 max pooling layers and only 1 fully connected layer.

Similar to the PilotNet architecture implementation described in the previous section, this architecture also includes a normalization layer used to normalize the input image between 0 and 1. On top of that, dropout layers, with a dropout rate of 20%, were also used in this architecture after the flatten layer and the fully connected layer. Finally, the linear function was also used as the activation function of the output layer.

As explained next, this architecture was only employed for driving on the OL and trained only with OL Dataset. The network has a total of 105493 trainable parameters, which is approximately 20% of the size of PilotNet. This size difference did not allow the JNet to effectively train on the remaining datasets.

The architecture of the network employed is illustrated in Figure 4.19, and the specification of the layers is shown in Table 4.5.

[4] https://keras.io/api/layers/core_layers/lambda/. [Last accessed in 03/06/2021]

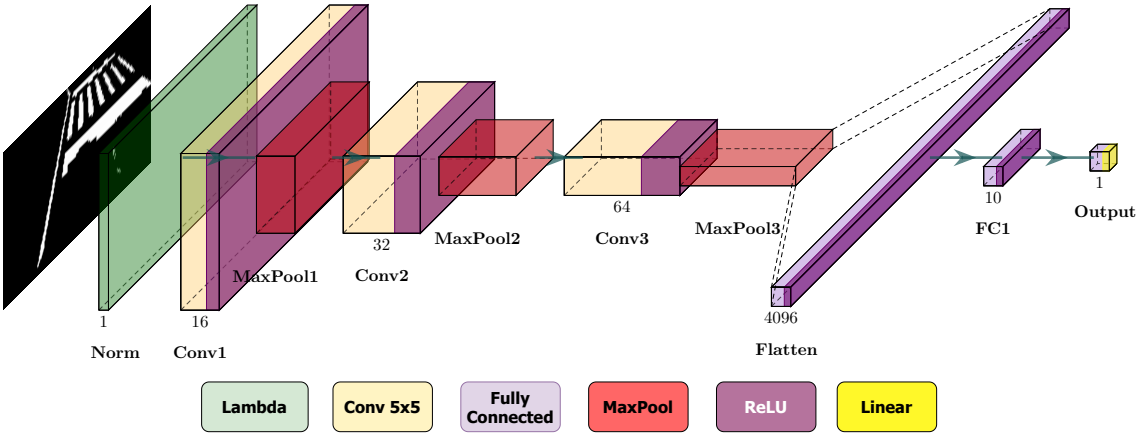


Figure 4.19: Visual representation of the JNet architecture.

Table 4.4: Parameters of PilotNet CNN architecture

Layer Name	Layer Type	Nº of Nodes	Nº of Filters	Filter Size	Stride	Padding	Activation Function
Norm	Lambda	65 x 160	1	65x160	1x1	No	-
Conv1	Convolutional	65 x 160	24	5x5	1x1	Zero	ReLU
MaxPool1	Max Pooling	32 x 80	24	2x2	1x1	No	-
Conv2	Convolutional	32 x 80	36	5x5	1x1	Zero	ReLU
MaxPool2	Max Pooling	16 x 40	36	2x2	1x1	No	-
Conv3	Convolutional	16 x 40	48	5x5	1x1	Zero	ReLU
MaxPool3	Max Pooling	8 x 20	48	2x2	1x1	No	-
Conv4	Convolutional	6 x 18	64	3x3	1x1	No	ReLU
Conv5	Convolutional	4 x 16	64	3x3	1x1	No	ReLU
Flatten	Flatten	4096	-	-	-	-	-
FC1	Fully Connected	100	-	-	-	-	ReLU
FC2	Fully Connected	50	-	-	-	-	ReLU
FC3	Fully Connected	10	-	-	-	-	ReLU
Output	Fully Connected	1	-	-	-	-	Linear

Table 4.5: Parameters of JNet CNN architecture.

Layer Name	Layer Type	Nº of Nodes	Nº of Filters	Filter Size	Stride	Padding	Activation Function
Norm	Lambda	65 x 160	1	65x160	1x1	No	-
Conv1	Convolutional	61 x 156	16	5x5	1x1	No	ReLU
MaxPool1	Max Pooling	30 x 78	16	2x2	1x1	No	-
Conv2	Convolutional	26 x 74	32	5x5	1x1	No	ReLU
MaxPool2	Max Pooling	13 x 37	32	2x2	1x1	No	-
Conv3	Convolutional	9 x 33	64	5x5	1x1	No	ReLU
MaxPool3	Max Pooling	4 x 16	64	2x2	1x1	No	-
Flatten	Flatten	4096	-	-	-	-	-
FC1	Fully Connected	10	-	-	-	-	ReLU
Output	Fully Connected	1	-	-	-	-	Linear

4.4 Model Training

To train the four models using each of the four datasets formed, they were divided into two parts. The first part, composed of 70% of the dataset, forms the training set, and the second part, composed of the remaining 30% of the dataset, forms the validation set. Table 4.6 gives the number of samples from the training set and the validation set for each of the datasets.

Table 4.6: Number of samples of the training and validation sets for each dataset.

Dataset	**Nº of Samples**	
	Training Set	Validation Set
OL	14668	6287
IL	12655	5424
OL^{-1}	6436	2759
IL^{-1}	6742	2890

The images from both sets, before being fed to the networks for training, are pre-processed with the operations described in Section 4.2.2. In addition, the image-angle pairs are shuffled to remove correlations between consecutive samples. In all training performed, the networks were trained over 25 epochs with a batch size of 32. As a loss function, the MSE was used (Equation 3.4) and to reduce the value of this function the Adam optimizer [22] was utilized with a learning rate of 1×10^{-4}. Early stopping was used as the regularization method. If after 7 epochs the validation loss does not decrease, the training is finished. At the end of each training, only the model with the set of weights that resulted in the lowest validation loss is saved and employed.

4.4.1 Training Results

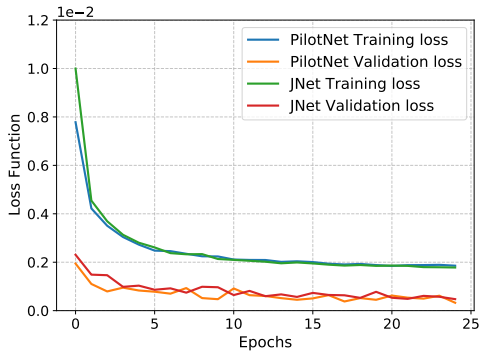
The plots in Figures 4.20a, 4.20b, 4.20c, 4.20d represent the training learning curves by showing the values of the loss functions of both the validation and training sets for each of the datasets built.

First, in all cases it is possible to observe that the functions decreased over the training epochs, which means that the models converged to the solution. This is also highlighted by the fact that in none of the cases Early Stopping was activated, which indicates that the validation loss was, most of the time, decreasing.

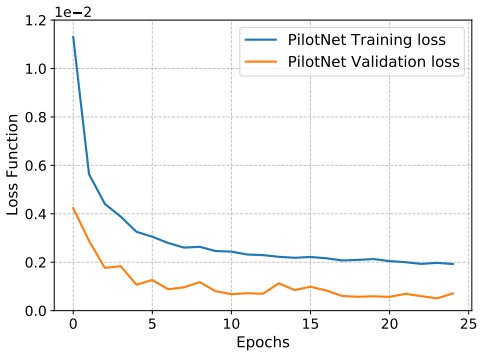
Secondly, at the end of 25 epochs both the validation loss and the training loss start to plateau and consequently the MSE starts to decrease with a very low rate, indicating that the models do not learn much more after that point.

In the situation in Figure 4.20a, where the two different architectures were trained, it is possible to observe a similar behavior in the learning curves of both architectures.

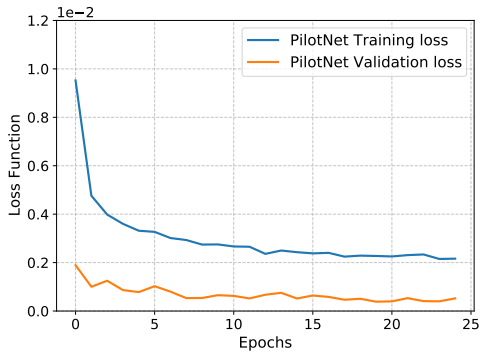
Finally, the training loss is always greater than the validation loss, which suggests that the models have not suffered from overfitting. This can be explained by the use of dropout layers.



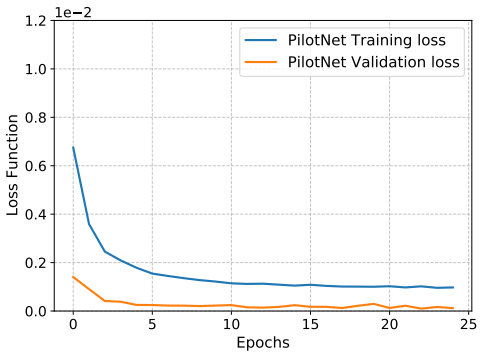
(a) PilotNet and JNet on OL Dataset.



(b) PilotNet on IL Dataset.



(c) PilotNet on OL^{-1} Dataset.



(d) PilotNet on IL^{-1} Dataset.

Figure 4.20: Training loss functions for each dataset.

4.5 Implementation details

This project was deployed on a PC equipped with Intel HD Graphics 4400 and AMD Hainan GPUs and a quad core Intel Core i7-4510U@2.00GHz CPU. Ubuntu 20.04.2 LTS was used as the operating system. Regarding software development, the programming languages used were C++ and Python.

For the implementation of the ML algorithms, the libraries Tensorflow[5](TF), Keras[6] and scikit-learn[7] were used. As for the image processing, the OpenCV[8] library was used.

For training the models, the Google Collaboratory (Colab)[9] cloud service was used. This Jupyter Notebook environment allows free access to Google's computational resources. These resources include GPU and TPU, often used in ML due to the computational power they offer for training ANNs. In addition, Google Colaboratory has the libraries mentioned above pre-installed and allows access to Google Drive, where the datasets were uploaded.

[5] https://www.tensorflow.org/. [Last accessed in 03/06/2021]
[6] https://keras.io/. [Last accessed in 03/06/2021]
[7] https://scikit-learn.org/. [Last accessed in 03/06/2021]
[8] https://opencv.org/. [Last accessed in 03/06/2021]
[9] https://colab.research.google.com/. [Last accessed in 03/06/2021]

For inference, Coral USB Accelerator[10] was used. This adds an EdgeTPU coprocessor to the PC by connecting to an USB port, providing high performance ML inferencing. However, this hardware only runs TF Lite[11] models that are fully 8 bit quantized. Therefore, the pre-trained models developed under Keras had to be converted to TF Lite using post training quantization with a representative dataset, which was also the one used for training. After this process, these models were compiled to run on the EdgeTPU using the EdgeTPU compiler. At the time of the implementation, these models were tested in the simulation environment in Gazebo and were included in the `/major_tracking` node of the ROS Architecture of the PDS, as explained in the next Section 4.6. Finally, for running the inference of the models deployed in the Coral Accelerator hardware, the TF Lite API and the EdgeTPU runtime were used.

4.6 Updated ROS Architecture

After training the models that drive the OL (OL Model), the IL (IL Model), the OL in opposite direction (OL^{-1} Model) and the IL in opposite direction (IL^{-1} Model), it remains to be seen where these are integrated into the ROS architecture of the previously PDS described in 2.3.

Since the CNNs receive as input the images from the tracking camera, the several trained models were integrated in the `/major_tracking` node. In this node, the images are pre-processed according to the operations described in Section 4.2.2 and then fed to the network, which infers the steering angle to be applied to the robot at that time instant and publishes it in the `/tracking_msg` topic. This is a modification relative to the PDS, since it no longer calculates the angle and distance references to be applied to the PID controller, but sends the steering angle directly.

Another modification in the system took place in the `/major_decision` node. This node continues to carry the intelligence of Major Alvega and gather the information coming from the other nodes of the system to make decisions. For this reason, the ability for this node to choose which of the 4 models it wants to run on the `/major_tracking` node was added. This was done by publishing the message of the desired model in the `/model_msg` topic that is subscribed by the `/major_tracking` node. In addition, this node adds to the steering angle message, posted in `/tracking_msg`, the velocity value to apply to the robot and publishes both in the `/major_msg` topic.

Finally, in this new control system there is no longer a PID controller which uses distance and angle references to the sidelines of the lane to calculate the angle to apply to the robot. This makes the `/major_control` node obsolete, so its only function is to take the steering angle and velocity values and translate them into ackermann steering commands published in the `/ackermann_cmd` topic.

Figure 4.21 depicts in red the changes made to nodes and topics in the ROS architecture of the PDS revealed earlier in Figure 2.8.

[10] https://coral.ai/products/accelerator/. [Last accessed in 03/06/2021]
[11] https://www.tensorflow.org/lite. [Last accessed in 03/06/2021]

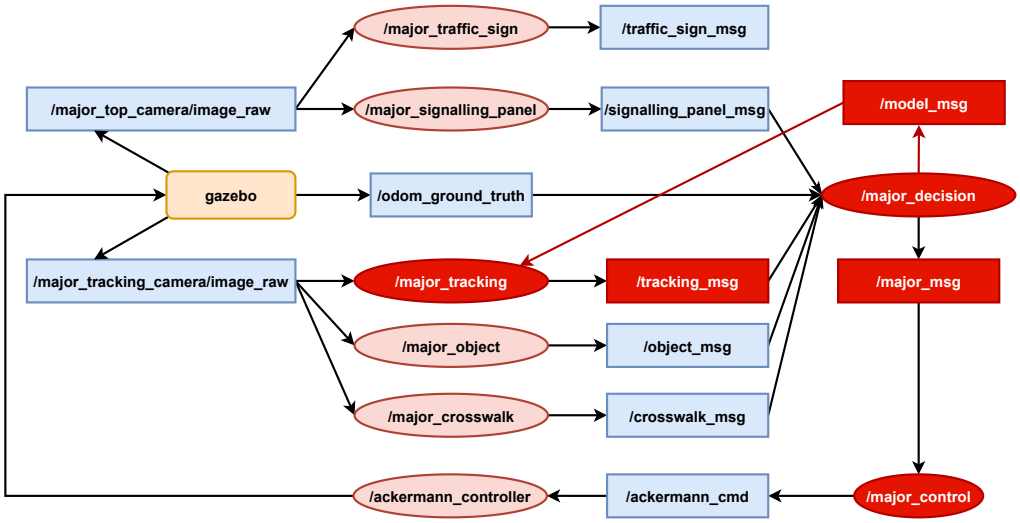


Figure 4.21: Major Alvega updated ROS architecture system.

Chapter 5

Experimental Results and Discussion

This chapter introduces, in Section 5.1, the experimental procedures taken to evaluate the proposed solution. The results of those experimental procedures are presented in Section 5.2 and discussed in Section 5.3.

5.1 Experimental procedures

To measure the effectiveness and quality of the proposed solution and in order to compare it with the PDS, an evaluation of the robots' driving performance in the simulation environment was undertaken. The metrics used for this evaluation are detailed in the next Section 5.1.1. As explained in Section 2.2.2.1, the track used for all tests was designed in SVG format, to ensure that the track effectively has a size of 16.7×6.95m and exactly fits all the dimensions stipulated in the ADC rules [6], including the width 0.75 meters for each lane.

The evaluations made in Sections 5.2.1 and 5.2.2 were performed with Major Alvega running a full lap along the track, using the OL and IL driving models, respectively. The same evaluation was done for the PDS in order to make a comparison between the two types of the robot controllers. Four different velocities ranging from 0.5 m/s to 2 m/s were tested, thereby covering the velocities at which the datasets were recorded, as well as testing the behaviour of the models with velocities above and below of those included in the datasets.

Furthermore, an evaluation was made in Section 5.2.3 for the cases where the robot performs the D2 challenge and needs to switch models in runtime at the middle of the route, according to the information given by the signalling panels. In this evaluation, it was tested if the robot's performance is affected by this model swapping.

Finally, in Section 5.2.4 the proposed algorithm that uses the four trained models (Figure 4.15) was tested on the performance of the obstacle avoidance D3 challenge. To do so, obstacles were placed in different positions in the second half of the track and the path taken by the robot was evaluated.

5.1.1 Evaluation Metrics

To evaluate the robot's driving performance in the simulation environment, three types of errors were measured. The first was the lateral error, which was given by the distance of the robot's rotation axis to the centre of the lane where it is located. The second was the orientation error, which was given by the relative angle between the robot orientation and the lane center. These errors are represented in Figure 5.1 as $E_{lateral}$ and E_{orient}, respectively. Even though in this figure, the reference point for the error calculations is depicted in the geometrical central of the robot, in this work the front wheels rotation axis was considered as the reference point.

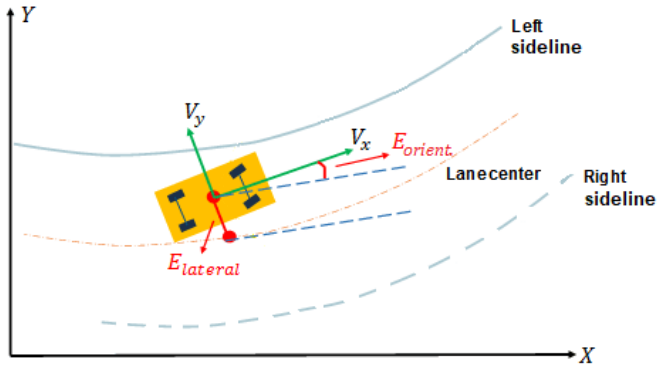


Figure 5.1: Lateral error ($E_{lateral}$) and orientation error (E_{orient}) representation. Adapted from [63].

For the sign of the lateral error, the following was assumed:

- If the rotation axis of the robot is distant from the lane centre in the direction of the continuous line, the sign is positive.
- If the rotation axis of the robot is distant from the lane centre in the direction of the broken line, the sign is negative.

Also, each lane of the track has a width of 75cm, whereas the robot has a width of 30cm. Therefore, one of the sidelines is crossed by the vehicle when an absolute error of 22.5cm[1] is reached. This value is therefore taken as a reference for the lateral error. In addition, the maximum and minimum error values are also taken to determine respectively whether the continuous line and or broken line have been transposed if above this reference value.

In order to quantify both of these errors over one full lap, the MSE, the Root Mean Squared Error (RMSE) and the Mean Absolute Value (MAE) were used as metrics:

$$MSE = \frac{1}{N}\sum_{i=1}^{N} e_i^2 \tag{5.1}$$

[1]Half of the lane width (37.5 cm) − Half of the robot width (15 cm)

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N} e_i^2} \tag{5.2}$$

$$MAE = \frac{1}{N}\sum_{i=1}^{N} |e_i| \tag{5.3}$$

where N is the number of samples collected over the entire path, and e_i is the error of the i^{th} sample.

The MAE and RMSE are expressed in the same units as the measurement, whereas MSE expresses these units squared.

In the cases where the robot failed to perform a full lap, the errors were quantified only for the performance on the first half lap.

The third and last error was based on the steering angles predicted by the models and applied to the robot. For such, the Mean Continuity Error (MCE) metric purposed by Chen et al. [64] was used:

$$MCE = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N-1} (a_{i+1} - a_i)^2} \tag{5.4}$$

where N is the number of frames, and a_i and a_{i+1} are the steering angles in radians applied to the robot for frame i and $i+1$, respectively.

This metric measures the fluctuations of steering angles, where a high MCE value indicates a large jumps between consecutive steering angle values or predictions, which leads to less stable and smooth driving.

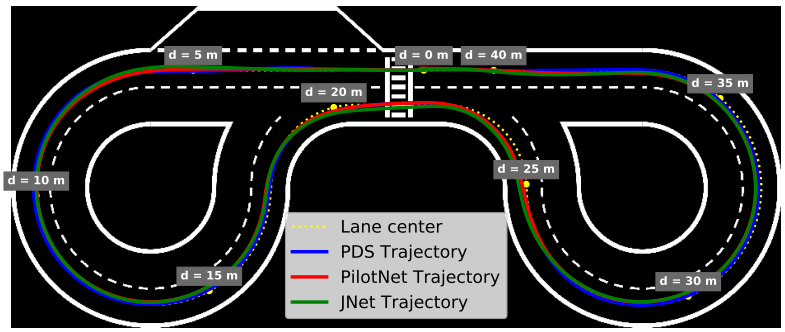
5.2 Results

5.2.1 Outer Lane models evaluation

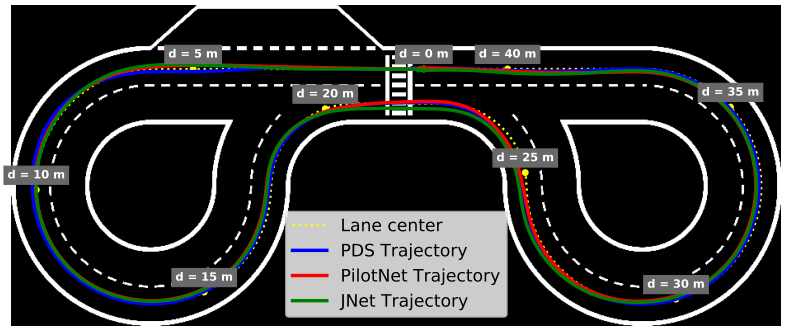
In this section, a comparison was established between the PDS and the proposed controller on the OL driving. For this purpose, an evaluation of both methods was made during the driving of one full lap, testing their performances for different test velocities. It should be remembered that for this lane, two different CNN architectures were implemented: the PilotNet and the JNet.

Figures 5.2a, 5.2b, 5.2c, 5.2d display the trajectories performed by the robot for those same velocities when using the PilotNet and JNet models and when using the PDS. These trajectories were compared with the lane centre, as the reference trajectory would be the one where the robot drives on the lane always at the same distance from both sidelines and orientated with them. As can be seen in Figure 5.2d, at the speed of 2m/s, the model using the JNet architecture was not able to perform a full lap at this speed because it deviated from the original path.

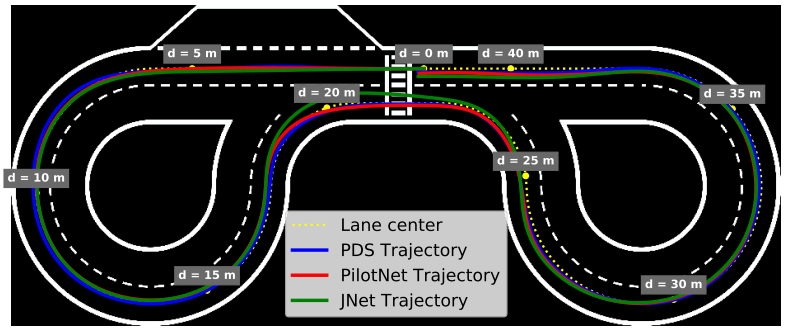
Based on these trajectories, the steering angle, lateral error and orientation error results were obtained and are presented in the following sections.



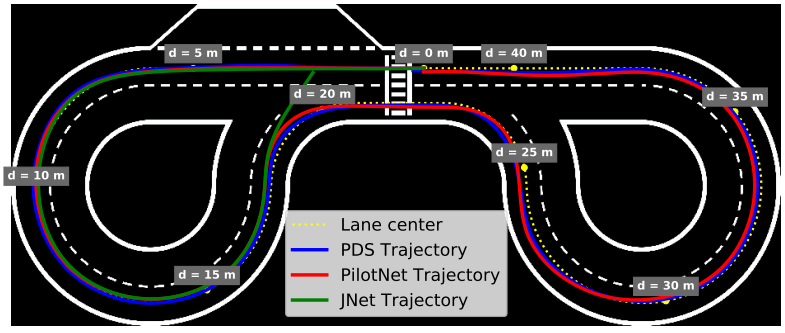
(a) Velocity = 0.5 m/s



(b) Velocity = 1 m/s



(c) Velocity = 1.5 m/s

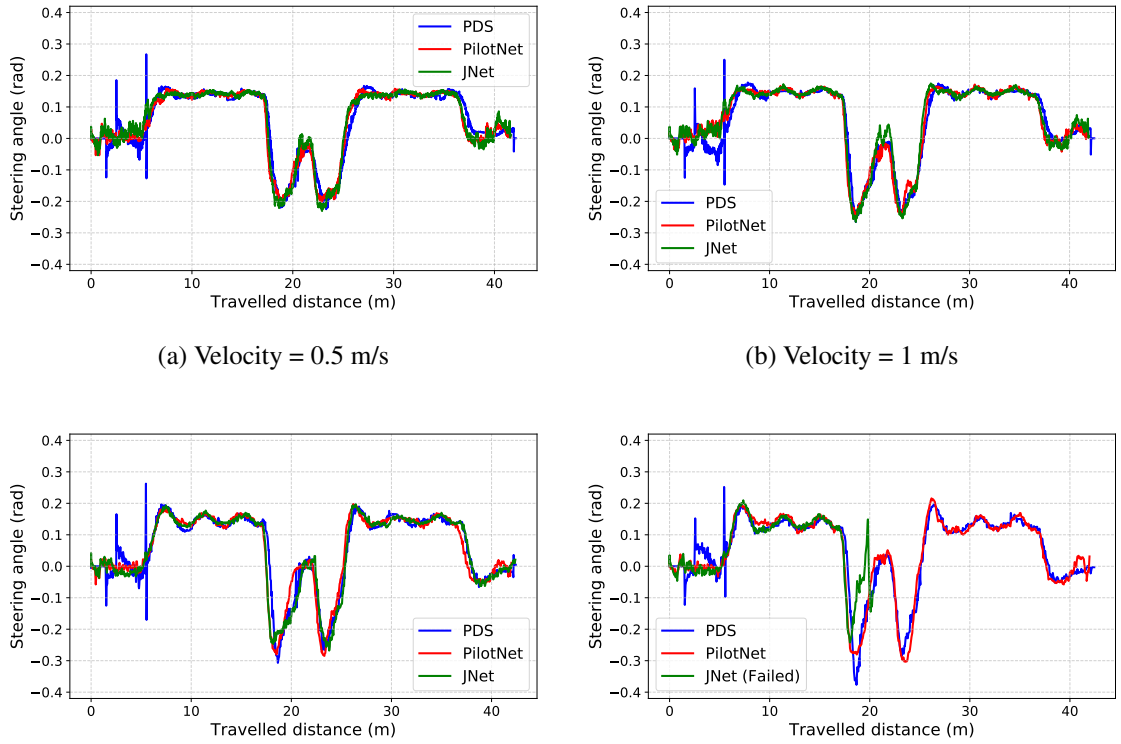


(d) Velocity = 2 m/s

Figure 5.2: Robot trajectories in the OL for the test velocities, using the PDS and the models trained with the OL Dataset.

5.2.1.1 Steering Angle results

Figures 5.3a, 5.3b, 5.3c, 5.3d display the plots of the steering angles applied to the robot in relation to the distance travelled on the OL of the track for each velocity. Furthermore, the data in Table 5.1 measures the MCE of the different models/systems and speeds during the whole route.



(a) Velocity = 0.5 m/s

(b) Velocity = 1 m/s

(c) Velocity = 1.5 m/s

(d) Velocity = 2 m/s

Figure 5.3: Steering angles applied during the performance of a full lap in the OL for the different test velocities.

Table 5.1: Evaluation comparison of the MCE of the steering angle in the OL performance.

Velocities (m/s)	Model	MCE (rad)
0.5	PDS	0.013
	PilotNet	**0.005**
	JNet	0.007
1	PDS	0.019
	PilotNet	**0.008**
	JNet	0.009
1.5	PDS	0.025
	PilotNet	**0.009**
	JNet	0.010
2	PDS	0.025
	PilotNet	**0.011**
	JNet *	0.017[1]

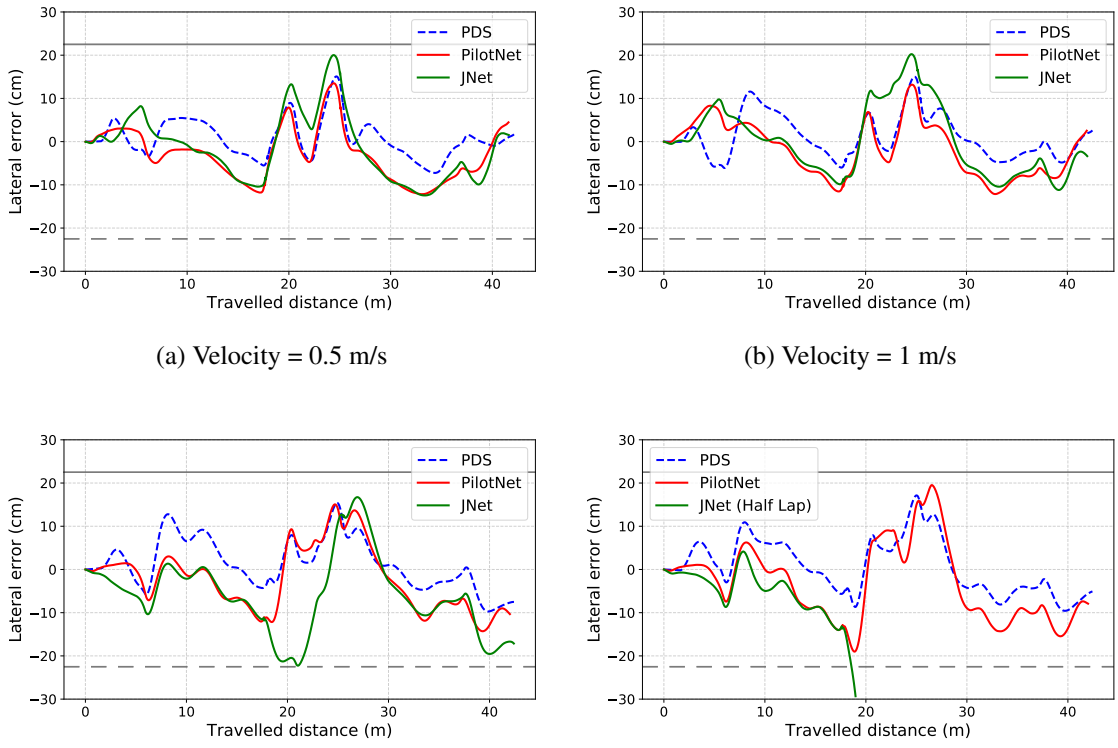
* Robot left the lane

[1] Measure on the 1st half of the track

These results reveal that the MCE was lower using any of the ML controller models compared to the PID controller of the PDS.

5.2.1.2 Lateral error results

Figures 5.4a, 5.4b, 5.4c, 5.4d presents the lateral error plots obtained over the distance travelled on the track by the OL for each velocity. Moreover, Table 5.2 complements the information in the plots, presenting the statistical evaluation of the lateral error for the different models/systems and speeds.



(a) Velocity = 0.5 m/s

(b) Velocity = 1 m/s

(c) Velocity = 1.5 m/s

(d) Velocity = 2 m/s

Figure 5.4: Lateral error plots of a full lap driving in the OL for the different test velocities.

Table 5.2: Evaluation comparison of the lateral error in the OL performance.

Velocities (m/s)	**Model**	**MAE (cm)**	**RMSE (cm)**	**Max (cm)**	**Min (cm)**	**MSE (cm²)**
0.5	PDS	**3.346**	**4.437**	15.087	**-7.250**	**19.684**
	PilotNet	5.496	6.642	**13.505**	-12.162	44.117
	JNet	6.247	7.743	20.025	-12.458	59.954
1	PDS	**3.948**	**5.103**	15.006	**-6.101**	**26.039**
	PilotNet	5.481	6.467	**13.176**	-12.138	41.820
	JNet	6.622	8.023	20.241	-11.196	64.371
1.5	PDS	**4.725**	**5.960**	15.414	**-9.718**	**35.525**
	PilotNet	6.243	7.597	**15.064**	-14.295	57.721
	JNet	8.524	10.571	16.732	-22.262	111.740
2	PDS	**5.336**	**6.393**	**17.133**	**-9.571**	**40.875**
	PilotNet	7.790	9.428	19.524	-19.008	88.890
	JNet *	5.873[1]	7.993[1]	4.154[1]	-29.331[1]	63.881[1]

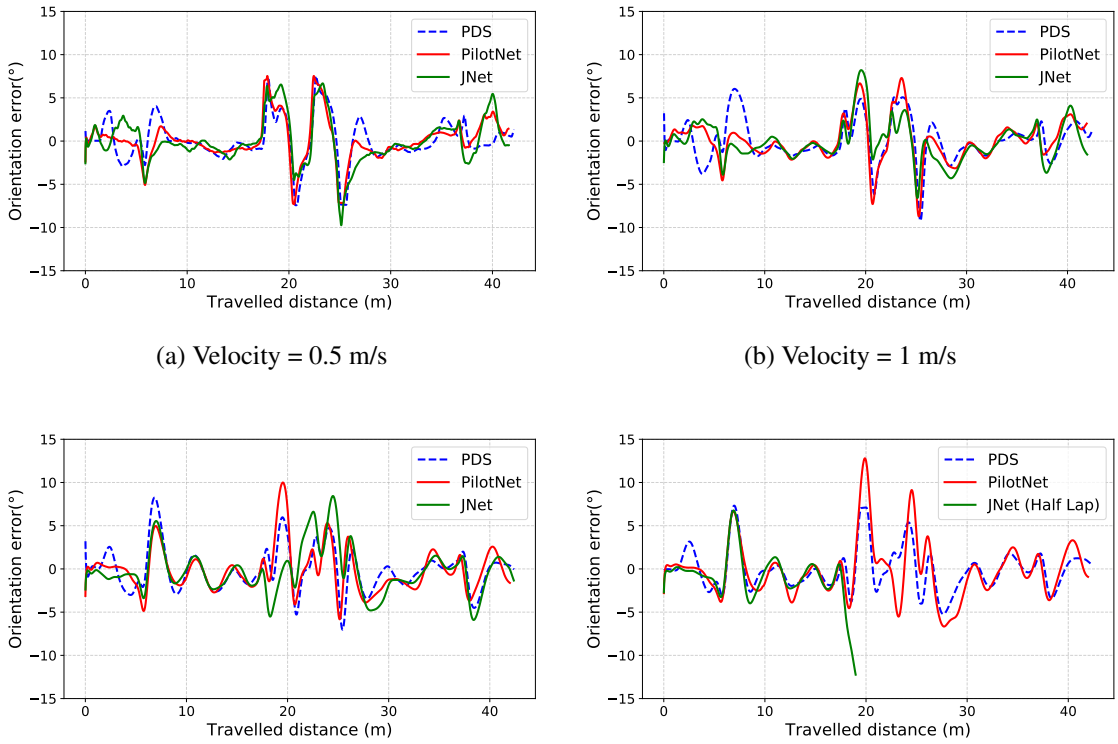
* Robot left the lane

[1] Measure on the 1st half of the track

Based on these results, it can be observed that the PDS presented the lowest MAE, the lowest RMSE and the lowest MSE for all velocities. However, for velocities below 2 m/s, the PilotNet presented the lowest maximal error. Furthermore, with the exception of JNet for the 2m/s test, in no case were the sidelines transposed as it is shown by the minimum and maximum error values below 22.5cm.

5.2.1.3 Orientation error results

Similarly to the lateral error, Figures 5.5a, 5.5b, 5.5c, 5.5d presents the plots of the orientation error obtained along the distance travelled on the track by the OL for each velocity. In addition, Table 5.3 completes the plots information, presenting the statistical evaluation of the orientation error for the different models/systems and speeds.



(a) Velocity = 0.5 m/s

(b) Velocity = 1 m/s

(c) Velocity = 1.5 m/s

(d) Velocity = 2 m/s

Figure 5.5: Orientation error plots of a full lap driving in the OL for the different test velocities.

Table 5.3: Evaluation comparison of the orientation error in the OL performance.

Velocities (m/s)	**Model**	**MAE (º)**	**RMSE (º)**	**Max (º)**	**Min (º)**	**MSE (º²)**
0.5	PDS	1.609	2.249	7.537	-7.490	5.060
	PilotNet	**1.511**	**2.236**	**7.536**	**-7.319**	**5.001**
	JNet	1.923	2.656	6.678	-9.741	7.054
1	PDS	1.775	2.395	**6.041**	-9.235	5.737
	PilotNet	1.731	2.412	7.301	-8.711	5.816
	JNet	**1.649**	**2.253**	8.208	**-6.587**	**5.078**
1.5	PDS	**1.801**	**2.443**	**8.256**	-7.024	**5.970**
	PilotNet	1.850	2.536	9.994	**-5.833**	6.432
	JNet	1.932	2.629	8.436	-5.909	6.914
2	PDS	**1.693**	**2.280**	**7.325**	**-5.235**	**5.199**
	PilotNet	2.257	3.258	12.799	-6.665	10.613
	JNet *	1.909[1]	2.954[1]	6.765[1]	-12.236[1]	8.725[1]

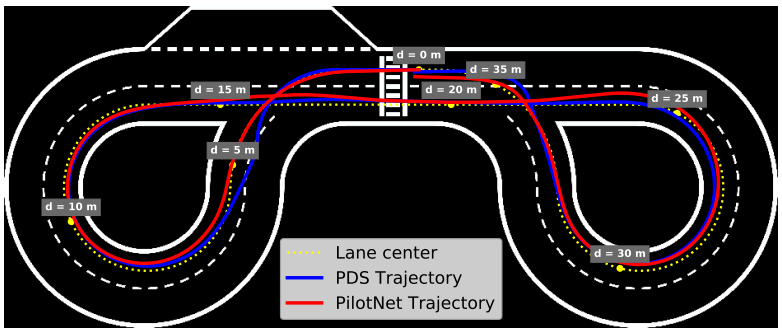
* Robot left the lane

[1] Measure on the 1st half of the track

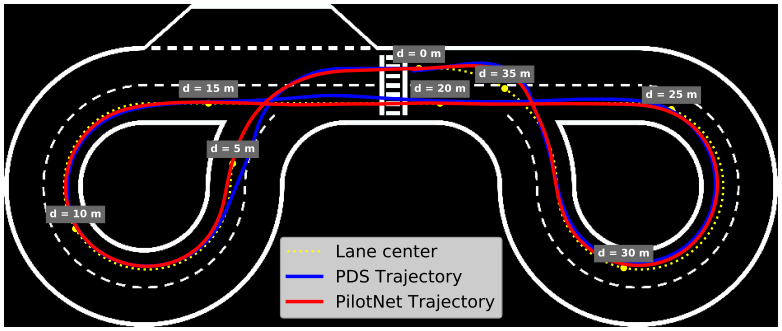
5.2.2 Inner Lane models evaluation

Similarly to the evaluation done in Section 5.2.1 for the OL driving, this section makes the same comparison between the PDS and the proposed controller, but here the robot drives in the IL. In this case, only the PilotNet architecture was trained with the IL dataset and employed in the

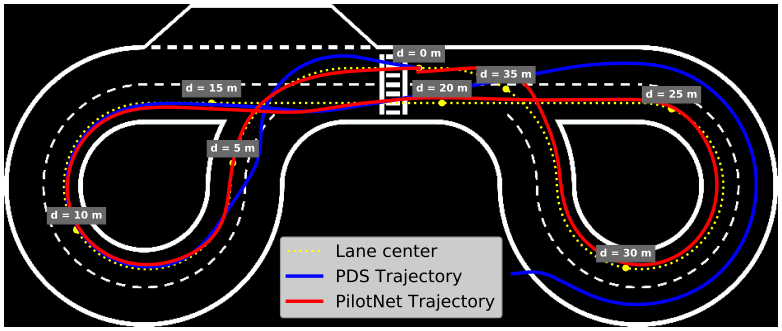
controller to drive in this lane. Figure 5.6a, 5.6b, 5.6c, 5.6d displays the trajectories performed by the robot when it uses PilotNet and PDS for the four test speeds. These trajectories are again compared with the center of the lane as a reference. For the cases depicted in Figures 5.6c and 5.6d, it can be seen that the PDS was unable to perform a full lap around the test track for the velocities of 1.5 m/s and 2m/s. In contrast, PilotNet performed a successful full lap in all the velocities undertaken. Based on these trajectories, the steering angle, lateral error and orientation error results were obtained and are presented in the following sections.



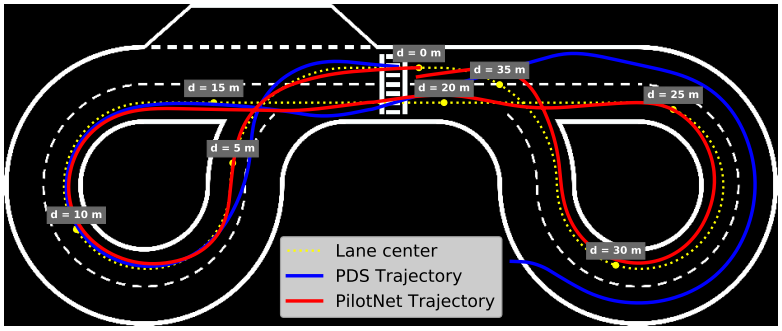
(a) Velocity = 0.5 m/s



(b) Velocity = 1 m/s



(c) Velocity = 1.5 m/s

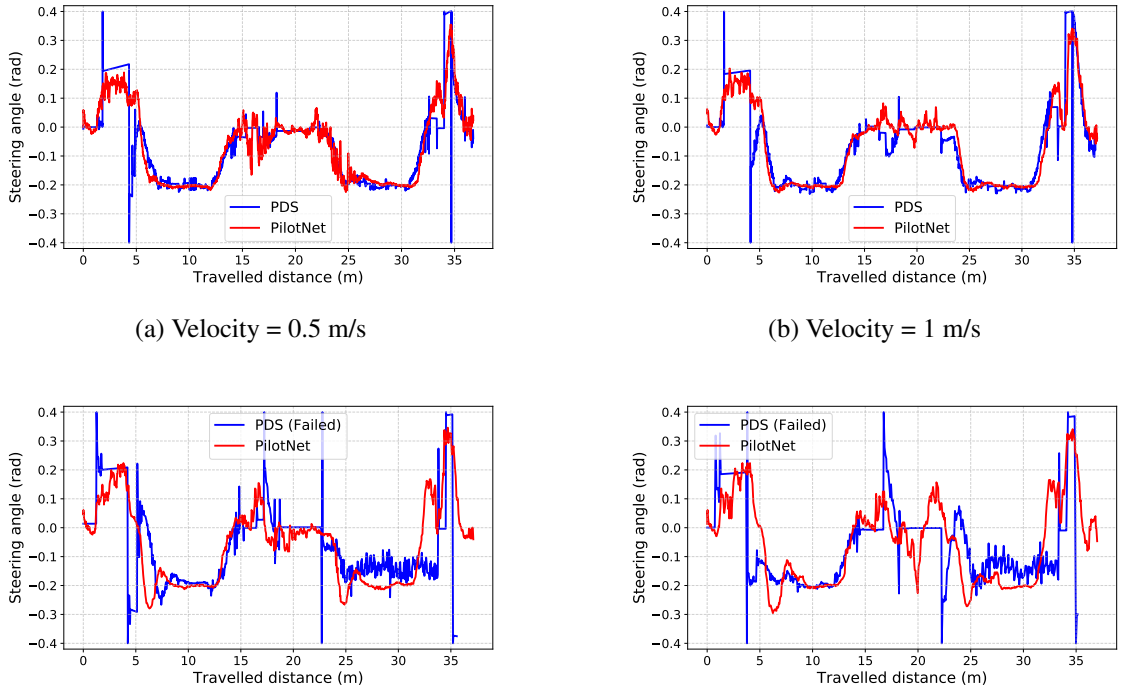


(d) Velocity = 2 m/s

Figure 5.6: Robot trajectories in the IL for the different test velocities, using the PDS and the models trained with the IL Dataset.

5.2.2.1 Steering angle results

Figures 5.7a, 5.7b, 5.7c, 5.7d plot the steering angles applied to the robot in relation to the distance travelled on the IL of the track for each velocity. Furthermore, the results in Table 5.4 evaluate the MCE of the different models/systems and velocities during the whole route in the IL.



(a) Velocity = 0.5 m/s

(b) Velocity = 1 m/s

(c) Velocity = 1.5 m/s

(d) Velocity = 2 m/s

Figure 5.7: Steering angles applied during the performance of a full lap in the IL for the different test velocities.

Table 5.4: Evaluation comparison of the MCE of the steering angle in the IL performance.

Velocities (m/s)	**Model**	**MCE (rad)**
0.5	PDS	0.032
	PilotNet	**0.015**
1	PDS	0.046
	PilotNet	**0.012**
1.5	PDS *	0.073[1]
	PilotNet	**0.017**
2	PDS *	0.080[1]
	PilotNet	**0.021**

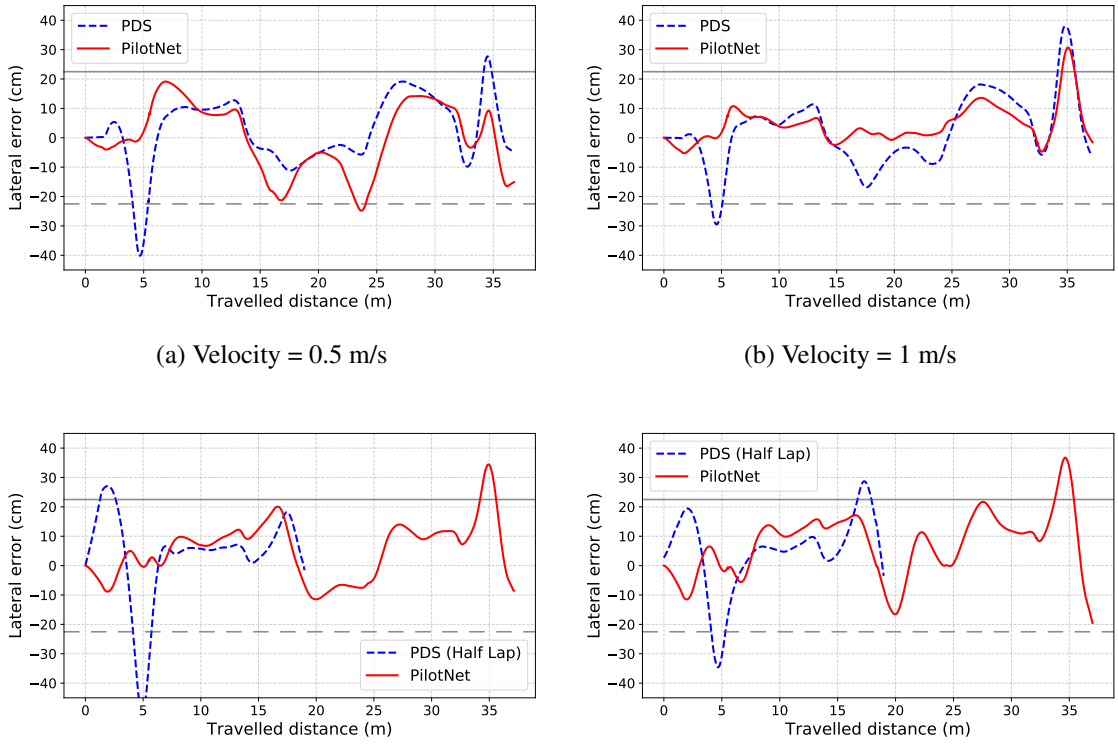
* Robot left the lane

[1] Measure on the 1st half of the track

Based on the obtained results, a lower MCE value of the proposed ML controller, compared to the previous controller, is once again verified on driving in the IL.

5.2.2.2 Lateral error results

Figure 5.8a, 5.8b, 5.8c, 5.8d presents the lateral error plots obtained over the distance travelled on the track by the IL for each velocity. Moreover, Table 5.5 complements the information in the plots, presenting the statistical evaluation of the lateral error for the PilotNet model and the PDS for the different test velocities.



(a) Velocity = 0.5 m/s

(b) Velocity = 1 m/s

(c) Velocity = 1.5 m/s

(d) Velocity = 2 m/s

Figure 5.8: Lateral error plots of a full lap driving in the IL for the different test velocities.

As noted in most of the plots, the maximum drift from the lane center occurs during the lane change maneuver that brings the robot back to the starting position in the second half of the track. This occurs near the 35 m of travelled distance. Although this value is above the reference value, no continuous line is crossed as this happens at the intersection. This deviation is nonetheless expected and the reason is explored in the discussion Section 5.3.2.

Table 5.5: Evaluation comparison of the lateral error in the IL performance.

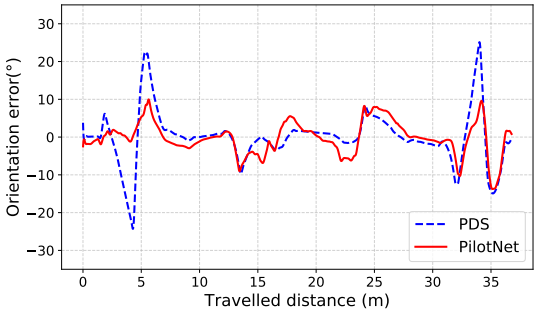
Velocities (m/s)	Model	MAE (cm)	RMSE (cm)	Max (cm)	Min (cm)	MSE (cm²)
0.5	PDS	**9.360**	11.935	27.716	-40.152	142.451
	PilotNet	9.677	**11.461**	**19.137**	**-24.776**	**131.345**
1.0	PDS	9.353	12.175	37.852	-29.506	148.232
	PilotNet	**5.394**	**7.777**	**30.719**	**-5.264**	**60.479**
1.5	PDS *	11.415[1]	15.886[1]	27.103[1]	-50.304[1]	252.379[1]
	PilotNet	**9.123**	**10.977**	**34.449**	**-11.476**	**120.487**
2.0	PDS *	10.902[1]	13.814[1]	28.687[1]	-34.626[1]	190.820[1]
	PilotNet	**10.730**	**12.891**	**36.774**	**-19.502**	**166.166**

* Robot left the lane

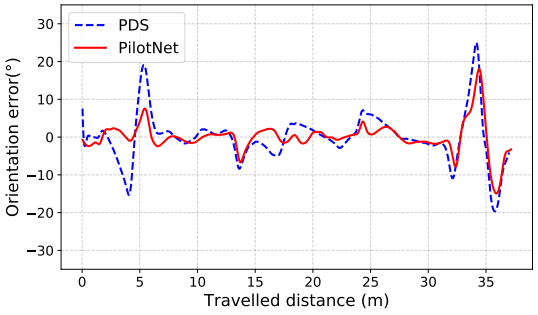
[1] Measure on the 1st half of the track

5.2.2.3 Orientation error results

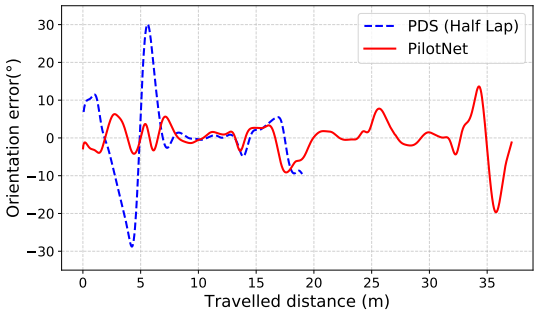
Lastly, Figures 5.9a, 5.9b, 5.9c, 5.9d presents the plots of the orientation error obtained along the distance travelled on the track in the IL. Moreover, Table 5.6 complements the information in the plots, presenting the statistical evaluation of the orientation error for the PilotNet model and the PDS for the different test velocities.



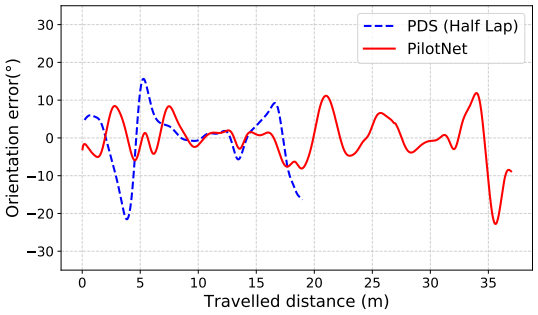
(a) Velocity = 0.5 m/s



(b) Velocity = 1 m/s



(c) Velocity = 1.5 m/s



(d) Velocity = 2 m/s

Figure 5.9: Orientation error plots of a full lap driving in the IL for the different test velocities.

Table 5.6: Evaluation comparison of the orientation error in the IL performance.

Velocities (m/s)	**Model**	**MAE (º)**	**RMSE (º)**	**Max (º)**	**Min (º)**	**MSE (º²)**
0.5	PDS	4.188	6.919	25.135	-24.308	47.879
	PilotNet	**3.206**	**4.352**	**9.982**	**-13.746**	**18.941**
1.0	PDS	4.147	6.454	24.972	-19.657	41.658
	PilotNet	**2.398**	**3.999**	**18.030**	**-14.959**	**15.988**
1.5	PDS *	6.501[1]	10.215[1]	30.099[1]	-28.784[1]	104.352[1]
	PilotNet	**3.218**	**4.757**	**13.633**	**-19.681**	**22.631**
2.0	PDS *	5.650[1]	7.749[1]	15.654[1]	-21.545[1]	60.047[1]
	PilotNet	**4.244**	**5.870**	**11.872**	**-22.781**	**34.454**

* Robot left the lane

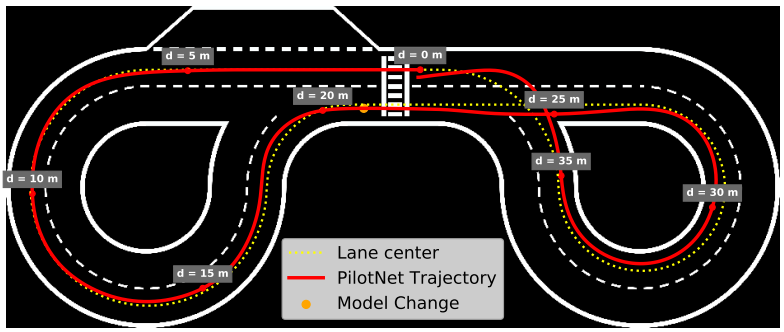
[1] Measure on the 1st half of the track

5.2.3 D2 Challenge evaluation

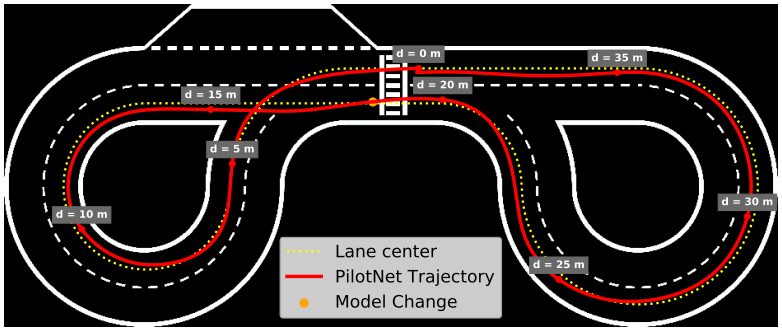
In this section the results are presented for the tests of the two specific cases of challenge D2, where a switch between the OL and IL models has to be performed. This exchange has to be conducted in runtime and at the middle of the route after detecting the signal displayed on the signalling panels. Since the system response speed is a crucial factor in this evaluation, the highest speed of all tests (2m/s) was used.

On the one hand, Figure 5.10a depicts the trajectory of a full lap of the robot when the "Follow straight ahead" indication was displayed on the two passages of the robot under the signalling panel. This triggered the activation of the OL model and the IL model in the first and second halves of the track, respectively. On the other hand, Figure 5.10b depicts the trajectory of a full lap of the robot when it was given "Follow to the Left" indication in the starting zone and "Follow to the Right" in the second passage under the signalling panel. This led to the opposite situation of the first one, where the IL model was activated in the first half of the track and the OL model in the other half. In both cases, the orange dot marks the point at which the two models switched in the controller.

Also, Figures 5.11a, 5.11b present the plots of the steering angles applied to the robot along the distance travelled in both cases. In this graphs, the moment when the models switched is marked by a vertical dashed line.

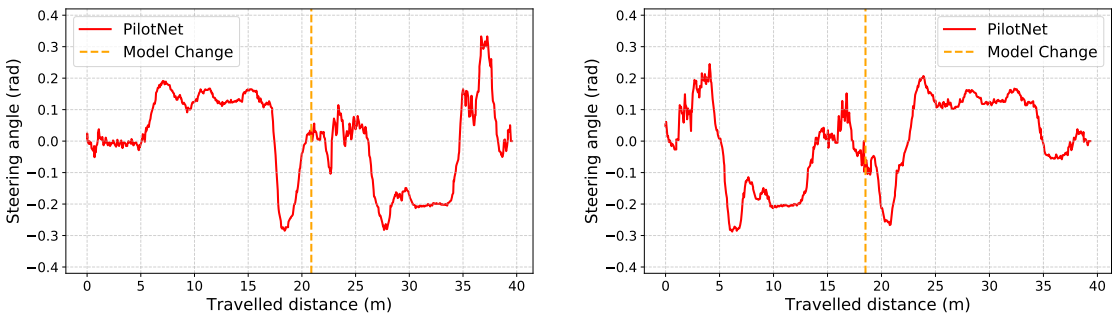


(a) OL Model → IL Model



(b) IL Model → OL Model

Figure 5.10: Robot trajectories for the two cases of the D2 challenge where model switching occurs.



(a) OL Model → IL Model

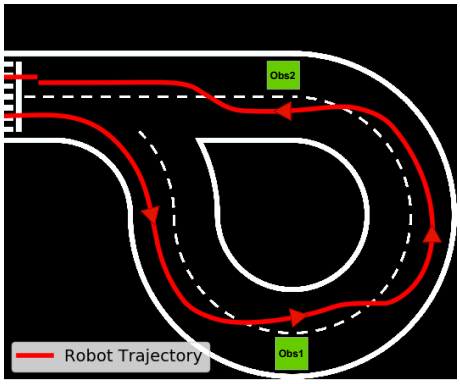
(b) IL Model → OL Model

Figure 5.11: Steering angles applied to the robot for the two cases of the D2 challenge where model switching occurs.

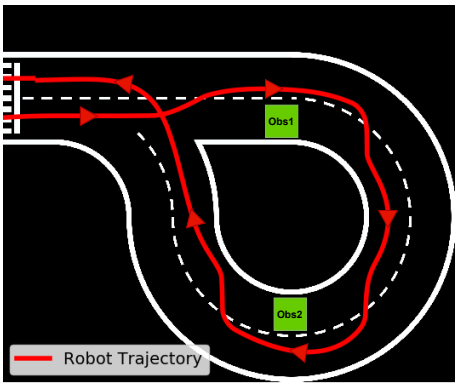
5.2.4 D3 Challenge evaluation

In this section the results of the tests performed for the obstacle avoidance challenge are presented. With these tests, it was intended to study the performance of the algorithm implemented for this challenge, when the obstacles are placed at different positions in the second half of the track. The algorithm was tested with the robot driving at a velocity of 1m/s on the lanes and performing the lane change maneuvers at a velocity of 0.5 m/s.

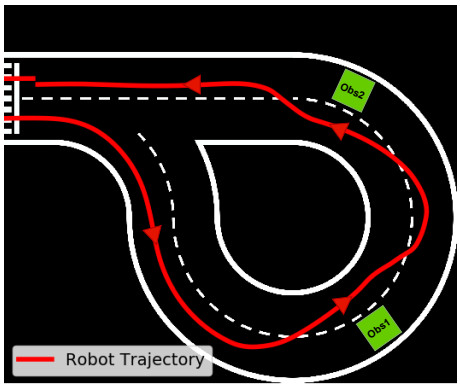
For this purpose, four different scenarios were tested with the placement of the two obstacles in positions similar to the ones that would occur in the ADC environment. In two of the scenarios, the obstacles were placed in the OL and in the other two, the obstacles were placed in the IL. These test scenarios are represented in Figures 5.12a, 5.12b, 5.12c,5.12d, as well as the trajectories performed by the robot.



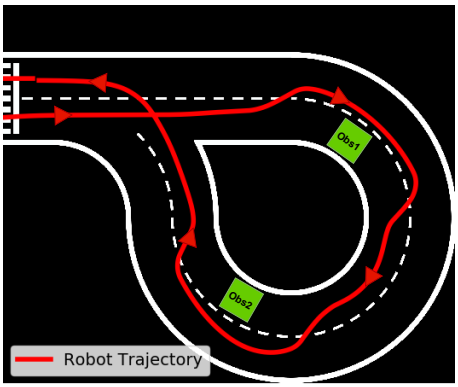
(a) OL test scenario 1



(b) IL test scenario 1



(c) OL test scenario 2

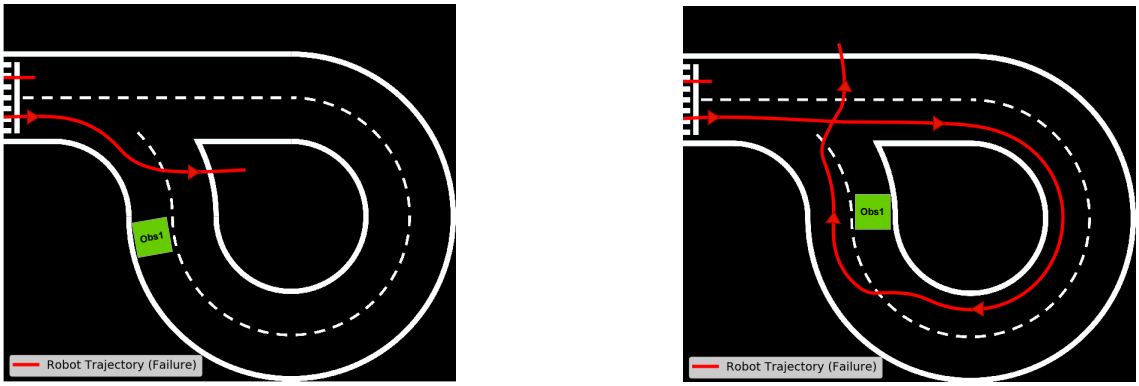


(d) IL test scenario 2

Figure 5.12: Obstacle avoidance challenge performance.

During the execution of the tests, some corner cases depicted in Figure 5.13 were identified. In these cases, the algorithm led the robot to leave the track, not being able to perform successfully a full lap. In the case depicted in Figure 5.13a, the obstacle was placed on the OL and the algorithm failed to change to the opposite lane after detecting the obstacle. For the case depicted

in Figure 5.13b, the obstacle was placed in the IL and, though the robot successfully circumvents the obstacle, it ultimately fails the maneuver of returning to the original lane.



(a) OL failure (b) IL failure

Figure 5.13: Corner cases where the obstacle avoidance algorithm has failed.

5.3 Discussion

5.3.1 Outer lane models evaluation

Firstly, by analysing the trajectories and steering angles (Figure 5.2 and 5.3, respectively), it can be confirmed that both CNN models have effectively learned the data from the OL dataset, as previously revealed by the training results for this dataset in Figure 4.20a. This is due to the fact that these models proved to be able to drive on the OL of the test track with a close approximation to the PDS for most speeds. The one exception was the case of JNet at the velocity of 2 m/s (Figure 5.2a and 5.3a). When performing the right turn near the zebra crosswalk, the network predicted a wrong angle around the 18 meter distance and, because of its high speed driving, the network was unable to correct its action afterwards, leading to the deviation of the robot from the lane. Relating this occurrence to the graphs of lateral error (Figure 5.4) and orientation error (Figure 5.5), it is easily observable that this zone of the zebra crosswalk, in the middle of the track, constitutes the most unstable zone of all the controllers, since the maximum values of both errors are reached here. This is mainly because this curve represents the sharpest curve in the OL, because the robot has to go from a left turn to a right turn in a relatively short distance. For higher speeds, the peak errors were even more significant, with the robot having to perform this same turn in a short time as well.

In addition to the steering angle plots in Figure 5.3, the results in Table 5.1 reveal an improvement in the smoothness of driving with the proposed controller compared to the previous PDS controller. This is evidenced due to the lower MCE for the PilotNet and JNet models compared to the PDS in all the tests performed, which led to the suppression of peaks and fluctuations in the steering angle signal. This effect was most noticeable during the first 6 meters travelled by the robot.

Based on the lateral error values in Table 5.2 and taking the reference values of 22.5cm, it can be concluded that the calculated errors for both systems are comparatively low and tolerable. This is because in no case did the trained models transpose any of the sidelines, with the exception of the JNet case for 2m/s. Regarding the orientation errors in Figure 5.5 and Table 5.3, low errors and low differences between the different models and velocities are also observed.

Overall, it can be concluded that the PDS continued to perform better than the trained models at driving on the OL. However, the difference in the performance is very negligible. By comparing the two implemented CNN architectures, PilotNet performed better than JNet since it performed a full lap at all speeds, with lower errors, smoother predictions and no failures.

Finally, it is worth reminding that this dataset contained only image-angle pairs in the real track, but it still performed well when tested in simulation environment. With this, it is possible to draw the same conclusions as [8], confirming that the simulator is quite realistic with only a small simulation-to-reality gap.

5.3.2 Inner lane model evaluation

By analysing the trajectories in Figure 5.6, it can be seen right from the start that PilotNet performed considerably better at driving in the IL when compared with the PDS.

Firstly, since this IL path has the particularities of left turns at the intersections in the first and second half of the track, a first analysis of these cases will be conducted. For the left turn in the first half of the track, the robot using PilotNet performed the maneuver as intended and close to the reference trajectory. However, by using the PDS, this maneuver was performed displaced from the reference trajectory and transversing the broken line. This difference in the performance of the maneuver is well evidenced in the graphs of lateral error (Figure 5.8) and orientation error (Figure 5.9), with the PDS reaching the maximum absolute error in these situations.

Regarding the left turn in the second half of the track, the robot has to perform such a maneuver that, after the circular curve, it returns to the starting position near the crosswalk in the OL. As is noted from the trajectory of both controllers, the robot always performs this maneuver displaced and disoriented from the lane centre. However, this behavior is intentional because it is desired that the robot returns to the OL and remains oriented parallel to it, as soon as possible and still with some distance behind the crosswalk. This distance allows the robot to have enough time to detect the signs from the signaling panels through the camera pointed upwards. For this reason, data from the robot performing this turn was intentionally included in the IL dataset and PilotNet learned to perform this turn as intended. This maneuver is again well evidenced in the graphs of lateral error (Figure 5.8) and orientation error (Figure 5.9). This time, it can be observed that, for most of the tests performed with PilotNet, the maximum error was reached during this maneuver.

Based on the steering angle plots in Figure 5.7, several observations are raised. The first one is once again related to left turns in the first and second half of the track. With the PDS, these maneuvers were performed using open-loop control by applying fixed steering angle values and in the transitions between the open loop and closed loop control severe overshooting was observed. With the PilotNet, although there were also some oscillations between predictions, this

overshoot was exponentially reduced and the steering was considerably smoother in comparison to the PDS. This fact is reflected in Table 5.4, which shows a lower MCE value for all the speed tests performed..

The second observation in this steering angle plot is related to the driving performance of the robot on the straight line. This line has the length of 10.6m and starts approximately at the distance travelled of 14m by the robot. In both cases, oscillations around the value 0 are verified, which caused some deviations from the centre of the lane when driving the robot on this part of the track. In the PDS case, these oscillations even led the robot to deviate from the lane for the velocities of 1.5 m/s and 2 m/s. The presence of these oscillations also in the PilotNet case may suggest that the network has learned this noise and has not generalized enough trained with PDS data. However the results for lateral error (Table 5.5) and orientation error (Table 5.9) revealed that these oscillations were not critical and that the proposed ML controller outperformed the PDS in driving performance in this lane.

Finally, it is important to recall that the IL dataset included data in both simulated and real environments. Furthermore, this dataset only included data with the Major Alvega driving only at a velocity of 0.85 m/s. The results of the lateral error and the orientation error demonstrate that the best performance of PilotNet on this lane occurs when a speed of 1 m/s is used. Being this test speed the closest to the one used in the datasets, the best performance in terms of lower lateral and orientation error is obviously given for this case.

5.3.3 D2 challenge evaluation

Based on the trajectories shown in Figure 5.10, it can be observed that no problem occurs in the driving performance of the robot when the switch between the two models is made. The plots in Figure 5.11 also confirms that no overshoot occurs in the steering angles applied to the robot at the transition point between the models. Only some oscillations of the IL model are observed which were already observable and aforementioned in the previous section.

The results of this evaluation and the previous individual evaluations of the OL and IL models, reveal that the proposed controller meets all the requirements of the D2 challenge of ADC, when implemented together with the `major_signalling_panel` and `major_crosswalk` nodes of the PDS.

5.3.4 D3 challenge evaluation

From the results in Figure 5.12, it can be seen that the obstacle avoidance algorithm (Figure 4.15) successfully performed challenge D3 for different test scenarios. These scenarios included instances where obstacles were placed in areas with different orientations in the second half of the track, namely the straight line and the circular curve areas.

Nevertheless, the results in Figure 5.13 reveal that there are some specific cases in which failed to perform as expected. In the case of Figure 5.13a, the failure occurs during the change maneuver to the opposite lane. The steering angle offset values were obtained empirically to cover

mostly the scenarios where obstacles are placed on the straight line and the regions in the circular curve where left turns occur. As this situation corresponded to a right turn, the offset value was overly high, positioning the robot transversely oriented to the lane. Consequently, the model of the opposite lane was unable to correct this orientation error and the robot deviated from the expected trajectory. Regarding the case of Figure 5.13b, although the robot has successfully performed the obstacle avoidance, the robot failed at the step of returning to the original lane. This was because this maneuver occurred near the region of the intersection, where the robot should have performed the left turn and not trying to return to the original lane. This step of the algorithm could then have been bypassed by switching directly to the IL model that would be capable of performing the left turn maneuver at the intersection.

It should be mentioned that obstacle placement such as in these two scenarios is unlikely to happen in the competition environment. However, these corner cases point out that the algorithm could be further improved by applying offset values adapted to the different orientations of the curves in the second half of the track.

Lastly, this algorithm revealed an alternative solution to perform challenge D3. For that, only the models trained exclusively for driving, the distance traveled based on odometry and the obstacle detection algorithm of the `/major_object` node of the PDS were used. Furthermore, to transfer the good behavior of the OL and IL models to the driving on the opposite direction lanes, it was only necessary to invert part of the OL and IL datasets, saving time implementing software to make the robot driving the lanes in the opposite direction. So, with this solution it was not necessary to build any exclusive dataset for obstacle avoidance.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The primary objective of this dissertation was to apply ML algorithms to control the navigation of the Major Alvega's robot that participates in the ADC.

The objective was accomplished, with the proposed ML controller being able to successfully perform a full lap around the track in both routes and while using different velocities. In addition, satisfactory results were observed regarding the deviation from the lane centre and the orientation error. In relation to the steering angle predictions by the models, low jumps between consecutive predictions were noted, which led to a smoother driving along the track, including in the turning maneuvers.

While integrating the proposed controller in the ROS architecture of the PDS, it was also possible to perform other challenges of the ADC competition. The first example was the performance of the D2 challenge, in which there were switches between the OL and IL models and the robot driving performance was not compromised. The second example was an extra feature of this dissertation and was related with the performance of the obstacle avoidance challenge. To perform this task it was only necessary to invert part of the data from the OL and IL datasets, train two more models with those inverted data and use the distance information obtained by odometry and the `major_object` from the PDS.

It is also possible to conclude that, by using the proposed controller, Major Alvega's navigation control system became significantly simpler by using end-to-end control. Firstly, the distance travelled obtained by odometry and the PID controller were no longer required for keeping the robot within the lane, since the models infer directly the correct steering angle. Secondly, the volume of image processing at the `major_tracking` node was also significantly reduced. This has occurred because the image processing steps for calculating the distance and angle between the robot's referential to the sidelines has been removed, and only a small amount of pre-processing was performed before feeding the images to the network.

The construction of the ADC real track is susceptible to human error, so its dimensions may not be exactly the ones defined by the rules. Tests in a simulation environment with the track distorted and with its dimensions both reduced and increased by 5% were also performed, to simulate this human error. In all tests, the proposed controller had a similar performance to the one that was manifested in the results, which led to the conclusion that the trained models are able to operate not only in the track with the exact dimensions, but also on a track where minor errors can happen.

It is important to mention that the hardware utilized for the inference of the models was the EdgeTPU in alternative to GPU. While utilizing this hardware, the models had to be quantized and therefore lightweighted. However, this device was still able to give real time inference on the control of the robot's navigation by predicting the proper steering angle with low latency and without degrading the model's effectiveness. Therefore, it was possible to implement these models and reach real time ML inferring in the control of the robot's navigation, using only a low cost hardware, suitable for embedded platforms and that offers low power consumption as well.

Finally, this project provided a successful case of applying ML to the navigation control of a robot with a steering mechanism identical to a real car and which is inserted in an environment very similar to a conventional human road. Indicating that these algorithms could be also effective in the future when incorporated into real self-driving cars.

6.2 Further Work

Despite the good performance of the controller in the navigation of the robot using the track in the simulation environment, it would also be interesting to observe how it would perform in the real track and robot.

Moreover, the current mode of operation of the controller discards temporal issues, since the models predict the steering angle simply by using the single image captured by the camera at that particular instant. However, driving is a continuous process in which past driving actions influence the steering angle at the current moment. Two possible approaches to consider time information would be: 1) use a stack of consecutive images as input, instead of just a single one; 2) use a LSTM module in the network architectures, in a similar way to the C-LSTM described in Section 3.2.4.

The obstacle avoidance algorithm was an attempt to perform the D3 challenge only by using the driving models. Its effectiveness is far from being perfect since it was unable to work on some areas of the track where obstacles were placed, because the offset was a fixed value, independent of the location on the track. As such, this algorithm still has room for improvement. A possible approach could be to implement RL methods for obstacle avoidance.

Finally, the application of ML-based algorithms proved to be very effective for controlling the robot's steering. These methods have the potential to be applied in other modules of Major Alvega, such as the detection of obstacles, signals from the signalling panels, traffic signs and the crosswalk.

References

[1] Defense Advanced Research Projects Agency. The Darpa Grand Challenge. URL: https://www.darpa.mil/about-us/timeline/-grand-challenge-for-autonomous-vehicles, 2016. [Last accessed: 22/06/2021].

[2] Sociedade Portuguesa de Robótica. Portuguese Robotics Open - Autonomous Driving Competition: 2019 edition. https://web.fe.up.pt/~robotica2019/index.php/en/conducao-autonoma-2, 2019. [Last accessed: 22/06/2021].

[3] Sociedade Portuguesa de Robótica. Portuguese Robotics Open 2019. URL: https://web.fe.up.pt/~robotica2019/index.php/en/, 2019. [Accessed 20/05/2021].

[4] V. Costa, P. Cebola, P. Tavares, V. Morais, and A. Sousa. Teaching Mobile Robotics Using the Autonomous Driving Simulator of the Portuguese Robotics Open. In *Robot 2019: Fourth Iberian Robotics Conference*, pages 455–466. Springer International Publishing, 2020.

[5] V. Costa. Autonomous Driving Simulator for the Portuguese Robotics Open. URL: https://github.com/ee09115/conde_simulator, 2020. [Accessed 21/05/2021].

[6] Sociedade Portuguesa de Robótica. Rules for Autonomous Driving. URL: https://web.fe.up.pt/~robotica2019/images/fnr2019_Autonomous_Driving.pdf, 2019. [Accessed 20/05/2021].

[7] V. Costa, R. Rossetti, and A. Sousa. Autonomous driving simulator for educational purposes. In *2016 11th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–5. IEEE, jun 2016.

[8] V. Costa, R. Rossetti, and A. Sousa. Simulator for Teaching Robotics, ROS and Autonomous Driving in a Competitive Mindset. *International Journal of Technology and Human Interaction*, 13(4):19–32, 2017.

[9] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, volume 3, 2009.

[10] V. Costa, P. Cebola, and A. Sousa, A.and Reis. Design of an Embedded Multi-Camera Vision System—A Case Study in Mobile Robotics. *Robotics*, 7:12, 2018.

[11] V. Costa. Projeto e Casos de Estudo em Robótica Educativa envolvendo Visão Tempo Real. Master's thesis, Universidade do Porto, 2015.

[12] W. McCulloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[13] Stanford CS231n. CS231n Convolutional Neural Networks for Visual Recognition. URL: https://cs231n.github.io/neural-networks-1/, 2021. [Last accessed: 16/06/2021].

[14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988.

[15] X. Glorot, A. Bordes, and Y Bengio. Deep sparse rectifier neural networks. *Journal of Machine Learning Research*, 15:315–323, 2011.

[16] B. Xu, N. Wang, T. Chen, and M. Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, 05 2015.

[17] D. A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (ELUs). *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, pages 1–14, 2016.

[18] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[19] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[20] C. Darken, J. Chang, and J. Moody. Learning rate schedules for faster stochastic gradient search. In *Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop*, pages 3–12, 1992.

[21] L. N. Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472, 2017.

[22] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[23] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[24] M. D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[25] T. Tieleman and G. Hinton. Lecture 6.5 - RMSProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[26] M. A. Nielsen. Neural networks and deep learning, 2018. URL http://neuralnetworksanddeeplearning.com/. [Last accessed: 16/06/2021].

[27] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[28] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, 2015.

[29] Y. Lecun, L. Bottou, Y. Bengio, and P. Ha. Gradient Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, pages 1–46, 1998.

[30] M. Bojarski, D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to end learning for self-driving cars. *CoRR*, 2016.

[31] W. Song and J. Cai. End-to-End Deep Neural Network for Automatic Speech Recognition. *CS224N Projects*, pages 1–8, 2015.

[32] MathWorks. Convolutional Neural Network - Matlab & Simulink. URL: https://www.mathworks.com/discovery/convolutional-neural-network-matlab.html, 2016. [Last accessed: 16/06/2021].

[33] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, 2013.

[34] C. Olah. Understanding LSTM Networks. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015. [Last accessed: 16/06/2021].

[35] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. doi: 10.1109/5.58337.

[36] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8): 1735–1780, 1997.

[37] R. Sharma, V. Agarwal, S. Sharma, and M. S Arya. An LSTM-Based Fake News Detection System Using Word Embeddings-Based Feature Extraction. In Simon Fong, Nilanjan Dey, and Amit Joshi, editors, *ICT Analysis and Applications*, pages 247–255, Singapore, 2021. Springer Singapore.

[38] F. Gers, J. Schmidhuber, and F. Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural computation*, 12:2451–2471, 2000.

[39] F. Gers and J. Schmidhuber. Recurrent nets that time and count. In *Proceedings of the International Joint Conference on Neural Networks*, volume 3, pages 189 – 194 vol.3, 2000.

[40] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[41] T. Glasmachers. Limits of end-to-end learning. *CoRR*, abs/1704.08305, 2017.

[42] S. Shalev-Shwartz, O. Shamir, and Sh. Shammah. Failures of gradient-based deep learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3067–3075, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[43] A. Singh, L. Yang, K. Hartikainen, Ch. Finn, and S. Levine. End-to-end robotic reinforcement learning without reward engineering. *CoRR*, abs/1904.07854, 2019.

[44] A. Tampuu, T. Matiisen, M. Semikin, D. Fishman, and N. Muhammad. A survey of end-to-end driving: Architectures and training methods. *IEEE Transactions on Neural Networks and Learning Systems*, page 1–21, 2020. doi: 10.1109/tnnls.2020.3043505.

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[46] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pages 1–14, 2015.

[47] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 07-12-June-2015, 2015.

[48] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[49] M. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, M. Hasan, B. C. Van Essen, A. A. S. Awwal, and V. K. Asari. A state-of-the-art survey on deep learning theory and architectures. *Electronics*, 8(3), 2019. doi: 10.3390/electronics8030292.

[50] D. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *NIPS*, 1988.

[51] M. Bojarski, A. Choromanska, K. Choromanski, B. Firner, L. D. Jackel, U. Muller, and K. Zieba. Visualbackprop: visualizing cnns for autonomous driving. *CoRR*, 2016.

[52] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry. How does batch normalization help optimization? *Advances in Neural Information Processing Systems*, 2018-December(NeurIPS): 2483–2493, 2018.

[53] M. G. Bechtel, E. McEllhiney, M. Kim, and H. Yun. DeepPicar: A low-cost deep neural network-based autonomous car. *Proceedings - 2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2018*, 2019.

[54] J. Rodrigues. Deep learning applied to Ackermann steering vehicle driving. Master's thesis, Universidade de Aveiro, 2018.

[55] J. Kocić, N. Jovičić, and V. Drndarević. An end-to-end deep neural network for autonomous driving designed for embedded automotive platforms. *Sensors (Basel, Switzerland)*, 19(9), May 2019.

[56] Z. Chen and X. Huang. End-To-end learning for lane keeping of self-driving cars. *IEEE Intelligent Vehicles Symposium, Proceedings*, pages 1856–1860, 2017.

[57] H. M. Eraqi, M. N. Moustafa, and J. Honer. End-to-end deep learning for steering autonomous vehicles considering temporal dependencies. *CoRR*, abs/1710.03804, 2017.

[58] E. Santana and G. Hotz. Learning a driving simulator. *CoRR*, abs/1608.01230, 2016.

[59] Y. Fu, D. K. Jha, Z. Zhang, Z. Yuan, and A. Ray. Neural network-based learning from demonstration of an autonomous ground robot. *Machines*, 2019.

[60] H. Harutyunyan, K. Reing, G. Ver Steeg, and A. Galstyan. Improving generalization by controlling label-noise information in neural network weights, 2020.

[61] J. J. Meyer. End-to-End Learning of Steering Wheel Angles for Autonomous Driving. Master's thesis, Freie Universität Berlin, Germany, 2019.

[62] U. Muller, J. Ben, E. Cosatto, B. Flepp, and Y. Cun. Off-road obstacle avoidance through end-to-end learning. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems*, volume 18. MIT Press, 2006.

[63] MathWorks. Lane Keeping Assist System. URL: https://www.mathworks.com/help/mpc/ref/lanekeepingassistsystem.html, 2021. [Accessed 21/06/2021].

[64] Y. Chen, P. Palanisamy, P. Mudalige, K. Muelling, and J. M. Dolan. Learning on-road visual control for self-driving vehicles with auxiliary tasks. *CoRR*, 2018.