Tuning Trajectory Following Controllers for Autonomous Cars Using Reinforcement Learning

Ana Vilaça Carrasco

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. João Fernando Cardoso Silva Sequeira

December 2021

Declaration

Abstract

Over the years, autonomous vehicles rapidly became the most popular research field of the automotive industry. In the past decade, Machine Learning (ML) has been one of the most popular approaches for solving the autonomous driving problem, more specifically Reinforcement Learning (RL) and Deep Learning (DL). When designing an autonomous vehicle, a common strategy involves using ML methods to tune the parameters of the vehicle's controller.

This thesis proposes the use of a *Q-Learning* algorithm to train an agent to tune the parameters of a trajectory controller. This controller is used on a system designed to perform two maneuvers and is also capable of avoiding frontal collisions and controlling velocity and steering angle range. A framework allowing a Q-learning based system to work with the CARLA simulator is introduced.

Initial tests were performed to the simulator to evaluate its performance. The results after the training demonstrate an appropriate implementation of the algorithm. A custom environment was created to test all the functions of the system, as well as the quality of the tuned gains. The results show the system performing all the functions as intended. They also suggest the gains were appropriately tuned, attesting to the success of the proposed algorithm.

Future works include real-world experimenting with the system, upgrading the Q-learning algorithm to a Double Q-learning, and adding a Neural Network for image processing of the environment.

Keywords

Machine Learning,Reinforcement Learning,Autonomous Driving Systems,Artificial Intelligence, Q-Learning

Resumo

Na última década, Machine Learning (ML), mais especificamente Reinforcement Learning (RL) e Deep Learning (DL), têm sido uma das abordagens mais populares para a resolução do problema da condução autónoma. Na concepção de um veículo autónomo uma estratégia comum envolve a utilização de métodos de ML para afinar os parâmetros do controlador do veículo.

Esta tese propõe a utilização de um algoritmo *Q-Learning* para ensinar um agente a afinar os parâmetros de um controlador de trajectória. Este controlador é utilizado num sistema concebido para efectuar duas manobras, ser capaz de evitar colisões frontais e controlar a velocidade e o limite do ângulo de direcção. É introduzida uma *framework* para um sistema baseado em Q-learning que trabalha com o simulador CARLA.

Começou-se por testar o desempenho do simulador. Os resultados após o treino do agente demonstram uma implementação adequada do algoritmo.

Foi criado um ambiente personalizado para testar todas as funções do sistema, bem como a qualidade dos ganhos afinados. Os resultados mostram a execução de todas as funções da forma pretendida e sugerem que os ganhos foram afinados de forma adequada, provando o sucesso do algoritmo proposto.

Trabalhos futuros incluem testar o sistema no mundo real, actualizar o algoritmo para um Double Q-learning, e adicionar uma Rede Neuronal para processamento de imagem do ambiente.

Palavras Chave

Aprendizagem Autónoma, Reinforcement Learning, Veículos Autónomos, Inteligência Artificial, Q-Learning

Contents

List of Figures

List of Tables

Acronyms

ADS	Autonomous Driving System
DL	Deep Learning
NN	Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
FCNN	Fully Connected Neural Network
SL	Supervised Learning
UL	Unsupervised Learning
RL	Reinforcement Learning
IRL	Inverse Reinforcement Learning
DNN	Deep Neural Network
LiDAR	Light Detection And Ranging
RADAR	Radio Detection And Ranging
ACC	Adaptive Cruise Control
PID	Proportional Integral Derivative
ML	Machine Learning
MSE	Mean Square Error
RMSE	Root Mean Square Error

Chapter 1

Introduction

This section introduces the fields of Autonomous Driving Systems (ADSs) and Machine Learning (ML), as well as applications of ML technologies in autonomous driving. The motivation and proposed work for this thesis is defined. A table of concise descriptions of the most relevant concepts addressed in this section is provided in the appendix A. The contributions of this work are addressed in the last section.

1.1 Motivation and proposed work

The initial idea for this thesis was to create a ML based system capable of automatizing the process of tuning the controller gains, implemented in a fully autonomous vehicle. Initially, the plan was to have one or more autonomous agents programmed to optimize the gains based on different characteristics of the environment - the curvature of the road, the number of obstacles around the vehicle, the type of street - or even based on variables like the trajectory errors, velocity or acceleration.

This thesis proposes a system with a modular architecture and a Reinforcement Learning agent for lateral and longitudinal control of a vehicle. This agent is trained for **offline tuning of the trajectory control gains**, to minimize the trajectory errors. The trajectory controller is a **proportional controlle** defined by **three gain values**, as expressed in equations (3.3)-(3.5).The Reinforcement Learning (RL) agent, using a discretized tabular **Q-learning algorithm**, is trained to fine-tune these gains to perform two maneuvers: **lane changing** and **roundabout navigation**. The system is also programmed to control the vehicle's velocity and steering angle range and to avoid frontal collisions. The **CARLA Simulator** [1] is used to train and test the proposed system. The simulator communicates with the system through a **ROS** [2] bridge.

The following are the goals of this project:

1. Create a Reinforcement Learning agent and, using a simple Q-learning algorithm, train it while performing the two maneuvers (lane changing and roundabout navigation) to tune the set of gains to each maneuver;
2. Build a module that can identify when to perform each maneuver and communicate with the RL agent to adjust the gains appropriately;
3. Build a safety module that controls the vehicle's velocity and steering angle range and avoids frontal collisions.

The contributions of this work are presented in section 1.6. Section 2.4 explains the thought process behind some of the decisions made by reviewing some architectures contained in the literature that inspired those decisions[1].

1.2 Autonomous Driving Systems (ADSs)

Although autonomous driving has been a research field since 1926 [3], it was only in the 1980s that Ernst Dickmanns [4] designed a vision-guided driverless van, which made a breakthrough in the area by introducing computer software to the field of autonomous driving. In the following decades, world-wide research in car automation grew exponentially, and by 1995 there were already cars operating with Neural Networks (NNs). In the early 2000s, **DARPA** (Defense Advanced Research Projects Agency) brought forward unmanned ADSs with the ability to navigate difficult off-road terrain and avoiding obstacles [5]. Nowadays, there is an ongoing race to achieve full autonomous driving, with **Waymo** (Google) and **Model S** (Tesla) in the lead, each betting on different architectural approaches. While better hardware can have a significant impact on the performance of the vehicle, it can be argued that taking the lead on the market mostly depends on the development of better software [6].

The SAE Standard J3016 [7], identifies **6 levels of driving automation**: **Level 0** is defined by a full-time performance by the human driver; **Level 1** is defined by having some driver assistance system of either steering or acceleration but expecting the human driver to perform all remaining tasks; On the **second level**, the human driver performs part of the dynamic driving task; On the **third level** the system performs the dynamic driving task; At **level 4** it is not expected of the human driver to intervene; At **level**

[1]The base code used for the work developed in this Thesis can be found in this GitHub repository: https://github.com/anavc97/Thesis-MEEC-IST-2021.git . Please contact the author to access the contents

5 the full-time performance is made by the system, under all circumstances and environments. As of today, the leading commercial vehicles have not reached the last level of automation.

Autonomous driving technologies have a vast range of applications, the most notorious being fully autonomous vehicles, but also other explored technologies like Vehicle-to-vehicle and Vehicle-to-Infrastructure [8]. Partially autonomous technologies can be found in many applications (Adaptive Cruise Control (ACC), Parallel Parking Assist). Massive implementation of fully autonomous vehicles can have countless advantages: avoiding accidents, reducing congestion, optimizing resources like energy, time and money, and a more Eco-friendly impact in the world.

1.3 Machine Learning (ML)

The history of Machine Learning starts with Warren McCulloch & Walter Pitts(1943) [9] and Donald Hebb (1949) [10], laying the grounds for the work on **Neural Networking**. In 1980 Kunihiko Fukushima introduced the Neocognitron, the first **Convolutional Neural Network (CNN)** architecture. Not long after the **Recurrent Neural Network (RNN)** (Michael I. Jordan, 1986) and the **Long Short-Term Memory (LSTM)** (Hochreiter & Schmidhuber, 1997) were born (table A.1).

Up until the late 80's, ML development was mostly focused on the **Deep Learning (DL)** area. However, another ML area known as **Reinforcement Learning (RL)** came along. The **Q-learning algorithm** is developed in 1989 (Christopher Watkins) [11], which greatly improves the practicality and feasibility of Reinforcement Learning, consolidating the presence of this area in the research field. The next years saw exponential growth, especially in the field of DL.

Designing an autonomous agent based on ML requires choosing the **learning strategy**, the **training algorithm** and the **Neural Network architecture** (for DL agents):

- **Learning Strategy**: Choosing the best learning strategy may depend on the input, output and available data and resources. In Machine Learning, most training algorithms fall into one of three categories: **Supervised Learning (SL)**, **Unsupervised Learning (UL)** (Table A.1) and **Reinforcement Learning (RL)**, the latter being where this work focuses on.

- **Training Algorithm**: The algorithm chosen to train the agent influences the performance of the agent and the training time. Moreover, choosing the values for the algorithm's hyper-parameters is usually a long process of trial-and-error [12]. In Reinforcement Learning, algorithms are broadly divided into three classes: **value-based**, **policy gradient** and **actor-critic** algorithms (Table A.1). SL problems can broadly be grouped into two types of problems: **classification** problems, which consist of predicting a discrete class label, and **regression** problems, which consist of predicting a continuous quantity.

- **Neural Network Architecture**: To design a DL agent, many types of neural network architectures are used, the most commonly seen in the field of ADSs being **CNN**, **RNN**, **LSTM** and Fully Connected Neural Network (FCNN) architectures (see chapter 2). While simpler networks offer faster results for less computational resources, more complex networks offer higher performance and robustness to the system.

Q-Learning Algorithm

In Reinforcement Learning, the autonomous agent learns to improve its performance at an assigned task by interacting with its environment - it learns what actions to perform in which situations in order to maximize a reward function [13]. A RL problem defines a **state space**, S, a **set of actions**, A, a **state transition probability matrix**, P and a **reward function**, $R_s(a)$, that associates a reward value with each state-action pair, (s, a). The RL agent is to learn a policy π which maximizes the rewards.

Q-learning [11] is a model-free RL algorithm, meaning it does not need to define a state transition probability matrix (generally used to model the system). A simple Q-learning application defines a table, named **Q-table** and shaped $[state, action]$, with an utility value for each pair. It initializes it randomly and then updates those utility values (known as the ***Q-values***) by greedily maximizing a **value function**. Equation (1.1) shows the new kind of value function proposed by this algorithm, called the **Q-function**:

$$Q^{new}(s_t, a_t) \longleftarrow (1-\alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q(s_{t+1}, a)), \tag{1.1}$$

where $Q(s_t, a_t)$ is the Q-value of the state-action pair before the update, α is the **learning rate** $(0 < \alpha \leq 1)$, $r_t = R(s_t, a_t)$ is the **expected reward value** associated with taking the action a_t in state the state s_t, γ is the **discount factor** and $\max_a Q(s_{t+1}, a)$ is the **estimate of the optimal future value**. This equation can be understood as: the **new Q-value** of the (s, a) pair is the **expected reward** for executing action a in state s plus **the best Q-value** it can produce in the new state, s', weighted by γ. In [11], the learning rate value is defined for each episode, α_n, where n is the n^{th} episode.

Learning Rate and Discount Factor

The **learning rate**, α, determines how much the new information acquired overrides the old information. A value of 0 will make the agent learn nothing from the new acquired information, while a factor of 1 will make the agent completely ignore prior knowledge in turn of the new information, making the learning process quicker (but not necessarily better).

The **discount factor**, γ, models how much the future rewards are worth comparing to the current reward, r_t. A value of 0 will make the algorithm ”short-sighted” by considering only the current rewards, while a factor near 1 will make the algorithm look for long-term high rewards. Generally, this value should be very close to 1.

Convergence Theorem (Watkins & Dayan) [11]

Given bounded rewards $|r_n| \leq R$, learning rates $0 \leq \alpha_n < 1$, and

$$\sum_{n=1}^{\infty} \alpha_{n^i(x,a)} = \infty \ , \ \sum_{n=1}^{\infty} [\alpha_{n^i(x,a)}]^2 < \infty \ , \forall x, \ a, \tag{1.2}$$

then $Q_n(x,a) \rightarrow Q^*(x,a)$ as $n \rightarrow \infty$, $\forall x$, a, with probability of 1.

Using the nomenclature of this thesis, x is the current state s, R is the reward function $R(s,a)$, Q^* is the Q values for an optimal policy, n is the number of episodes and $n^i(x,a)$ represents the index of the i^{th} time that action a is tried in state x. This theorem defines the following conditions for convergence:

1. The reward, r_t is drawn from the reward function $R(s,a)$;

2. The learning rates of each episode, α_n, must satisfy the two properties of equation (1.2);

3. The (s,a) are visited infinitely often, i.e., $n \rightarrow \infty$;

If these conditions are met, the function converges to the optimal solution, $Q^*(s,a)$ with a probability of 1.

Exploration vs. Exploitation

The training process can be divided into two stages, the **exploration phase**, when the algorithm is mostly taking random actions to ”explore” the environment, and the **exploitation phase**, when the algorithm prioritizes past knowledge and always picks the best possible action for the current state. This work uses the **ϵ-greedy policy** to balance exploration and exploitation:

$$a_t = \begin{cases} \arg\max Q(s_t) & \text{with probability} \quad 1-\epsilon \\ \text{random action} & \text{with probability} \quad \epsilon \end{cases} \tag{1.3}$$

where ϵ represents the probability of choosing a random action, i.e., exploring. The action selection policy is used with an ϵ value that decays over time.

The Q-learning algorithm implemented is explained in more detail in section 3.3.1.

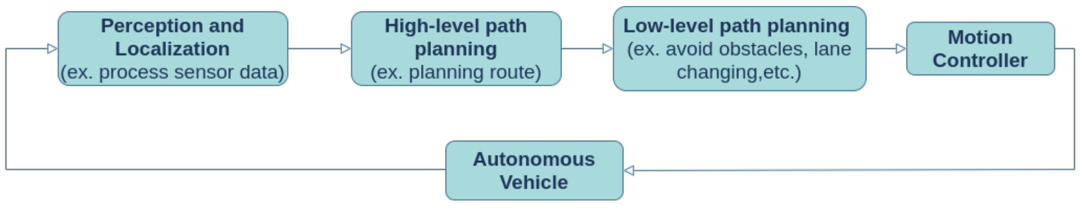
1.4 Machine Learning in Autonomous Driving Systems

Since the early 1980s, Machine Learning-based architectures have been one of the most popular approaches when designing ADSs. One of the first to research and implement Neural Networks in an ADS was Carnegie Mellon University, with **Navlab** [14], in 1984-1986. Pomerleau and his team presented **ALVINN**(1989) [15], a simple system with an end-to-end architecture that could successfully follow a road. Since then, projects like **DARPA Grand Challenges** brought forward multiple autonomous vehicles and, in 2007, fully autonomous vehicles were driving successfully for miles.

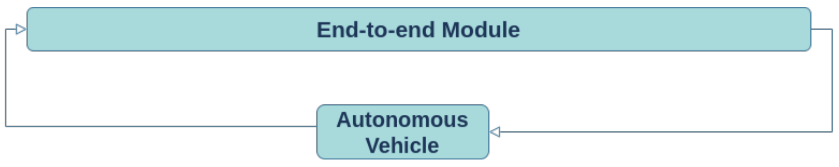
Machine Learning offers many benefits for the field of ADSs: self-optimisation based on collected data and new environments, the ability to specify the desired behaviour of the system, and increased generalisation capacity. Machine Learning, especially Deep Learning, has become the primary interest of the autonomous vehicle field.

1.4.1 Architecture Approaches

There are two different approaches when designing the architecture of an ADS: a **modular architecture**, with a perception-planning-control pipeline, or an **end-to-end learning architecture**, where the sensor data is directly mapped to a control output.



(a) Modular Architecture



(b) End-to-end Architecture

Figure 1.1: Generic architecture approaches

Modular architecture (figure 1.1(a)) usually consists of a perception module, a high-level and low-level path planning module, and a control module: the **perception module** is responsible for gathering the sensory information, mapping the surroundings of the car and localizing it; The **high-level path plan-**

ner consists of obtaining a reference path between the origin and the destination; the **low-level path planner** operates more locally than the previous module, such as to calculate trajectories around obstacles in the road; The **control module** is responsible for communicating with the autonomous vehicle, providing the appropriate control actions, like desired steering angle, velocity, or brake. Architectures based on modules separate the challenging tasks of autonomous driving into simpler and smaller problems, offering the advantage of expertly crafting the individual modules, either with ML or more classical methodologies, but also integrating functions on top of these modules - a redundant but reliable design [16].

An **end-to-end learning architecture** (figure 1.1(b)) is significantly simpler, with only one module, generally with a ML design, connecting directly the perception system with the autonomous vehicle. End-to-end architectures have been shown to improve the performance significantly in several robotic tasks [17] and do not require any intermediate engineered modules. The optimization offered by this approach directly mimics the performance demonstrated by the human drivers, a desirable feature. However, an architecture only based on ML lacks transparency on a concerning level: it is very difficult to understand how the ML agent makes decisions, and thus very difficult to ensure a safe and desired behaviour (this problem is known as the **"black box"** problem). Applying machine learning methodologies to only part of the modules and combining them with other classical methodologies gives the opportunity to ensure some level of safety and apply more restrictions to the behaviour of the agent. This is the main advantage explored in this thesis.

1.4.2 Partial and full vehicle control

Over the years, researchers have applied machine learning techniques to three different types of controls: **lateral control**, **longitudinal control**, and **full vehicle control**, where multiple tasks can be performed.

Lateral control consists of controlling the position of the vehicle on the lane, while longitudinal control consists of controlling the distance to the cars in front and also avoid rear-end collisions. Although generally focusing on one of the two types of control, more recent research explores the simultaneous lateral and longitudinal control. These three types of controls face different challenges, generally differing in type of learning strategy, Neural Network architecture, and algorithms used (see 2.2.4).

Different vehicle controls perform different tasks: longitudinal control performs tasks like maintaining a certain distance from the vehicle ahead, collision avoidance, safety and comfort balance, and emergency braking maneuvers. Lateral control tasks include road following, navigation around obstacles, and lane change maneuvers. Combining multiple tasks is explored when researching full vehicle control, which translates in goals like predicting desired steering angle and acceleration simultaneously, mitigating collisions or navigating through busy streets.

1.5 Organization of the Document

The structure of this thesis is the following: The two key topics of this study, ADS and ML, as well as the intersection of these two, are introduced in **chapter** 1. It also explains the motivation and describes the contributions of this work. **Chapter** 2 reviews the most important technologies of these areas, focusing on ML applied to the ADS industry. It also includes a section outlining the most common challenges to constructing an AI-based autonomous vehicle architecture. The chapter ends with a look at some architectural examples that influenced the choices made for this project. In **chapter** 3, the system's architecture is laid out. It is then broken down into modules, each of which described in greater detail. **Chapter** 4 describes the tests performed to the system and discusses the results from those tests. Finally, the conclusions of this work, as well as the proposed future work are presented in **chapter** 5.

1.6 Contributions

These are this work's contributions:

- Propose an offline approach to gain tuning: instead of tuning the gains while driving the vehicle through live updates of the Q-table values, this thesis proposes updating the Q-table and tuning the gains every time a specific maneuver is performed by the vehicle - the data is collected and processed afterwards, and the gains adjusted accordingly. This would mean that the gain tuning would occur in an extended period of time, rather than at each iteration. Offline tuning avoids possible sudden gain changes while driving, which can cause safety issues. This approach also allows the user to tune the gains to specific environments inside a city, which can be beneficial to use in well-known dangerous zones;
- Test the efficacy of a simplified algorithm: demonstrate that a Q-learning algorithm with limited state-action space and constricted combination of gains can still produce satisfying results. A simplified algorithm brings advantages like: code errors are less likely, easier to troubleshoot, easier to implement in a project with limited computational resources, and easier transition to a physical system;
- Integrate a Q-learning agent with the CARLA simulation environment;

Chapter 2

Literature Review

This chapter reviews the **latest technologies on the ADS and ML fields**, including a review of the relevant hardware used in autonomous vehicles and a review of the **most popular and common Machine Learning applications** in this field. The third section discusses the **most common challenges** faced when designing an autonomous driving system. The last section presents some of the **related work** that inspired the proposed architecture.

2.1 Sensors

The **perception system**, an essential part of an ADS, is made up of a set of sensors that use fusion techniques to combine the gathered information about the environment [18]. The general functioning of an ADS is heavily based on the perception of its surroundings, hence the perception system has a significant impact on the system's performance. The quality of the perception system is a key factor for commercial applications of autonomous driving technology in the real world.

The sensors that usually make up perception systems are: **Ultrasonic**, **RADAR**, **LiDAR** and **cameras**. **Sensor fusion** manages the advantages and disadvantages of each sensor to improve the quality of the data perceived. The list below presents a summary of how these sensors work, as well as the advantages and disadvantages of their application in the context of ADSs.

- **Ultrasonic:** They use sonic waves to measure the distance to an object. In vehicles, they are especially used in parking systems or other short-distance measurements at low velocities. They are cheap sensors that work well with any material and also in dusty or humid environments. However, they tend to produce false positives when bouncing and have blind zones in the measurements, which can lead to safety risks in some situations.

- **Radio Detection And Ranging (RADAR):** Measure the distance between the emitter and other objects by calculating the time between the emission of a radio signal and the received echo. RADARs detect short-range targets in a wide range around the vehicle, with high accuracy, providing information about distance, direction and speed of the targets. RADARs are especially good in all weather conditions. These two particularities make them significantly improve safety and reduce human decision-making burdens. On the other hand, they can produce false positives due to bouncing of the emitted signal, have reduced individual Field of View and are known for a lack of precision.
- **Light Detection And Ranging (LiDAR):** Measure the time between a pulsed infrared light emission from a laser diode until its reception by an emitter. They are known for their high resolution and great accuracy in perception of the environment, even in the dark. They are usually used in Adaptive Cruise Control, object identification and avoidance or simply 3D mapping. These are very expensive sensors that are affected by weather conditions such as rain, snow, fog or dust due to the diffraction of light in these conditions. They also reduce their operating range detection depending on the reflectivity of the targeted objects.
- **Cameras:** They are the most commonly used sensors in ADS perception systems, with the purpose of obtaining information about the surroundings with high-quality colour information and high resolution. They are especially well suited for object recognition and are also very cost-efficient. The most common cameras, known as visible cameras (for example, RGB) are highly affected by variations in lighting conditions, rain, snow, or fog, do not work in the dark and lack depth perception. In addition, high-resolution images are highly dimensional and compact a lot of useless data, which usually requires the implementation of techniques that extract only useful information.

2.1.1 Sensor Fusion

The goal of **sensor fusion** is to improve the measurement of two or more sensor data by applying redundant data measurements and combining them in an algorithm. There are several types of fusion methods and algorithms, either based on Gaussian filters, probabilistic inference or Machine Learning techniques. It is essential to obtain the best possible performance for the perception system. For instance, RGB cameras usually need to be complemented with other sensors that work better in the dark or with rain, to compensate for their big dependence on lighting conditions. Another problem worth considering when implementing sensors in ADS is the situation where one or more of the sensors malfunctions or is compromised. Having multiple sensors of the same type ensures that the information is not lost in this situation, or it can be compensated by other types of sensors through fusion methods.

In the ADS industry, there is little consensus on what should be the approach to a sensor system implementation: camera based or LiDAR based [19]. Nevertheless, there is no argument against having multiple, different sensors working together.

2.2 Machine Learning applications in Autonomous Vehicles

Three domains can be distinguished when designing an intelligent agent: **learning strategies**, **Neural Network architecture** and **training algorithm**[1].

2.2.1 Learning Strategies - Reinforcement vs. Supervised Learning

Reinforcement Learning is distinguished for the self-improvement and generalisation capabilities it offers: systems operating with RL are particularly good at learning from past experience and adapting to new environments. This strategy is suitable for typical time-sequential problems, where the completion of a task involves a sequence of actions that can impact each other, which is the case for autonomous driving. Unlike SL, it does not require a **labeled dataset** and instead offers the possibility of explicitly defining a desired behaviour for the agent (reward function). This may be an advantage or a disadvantage, depending on the complexity of defining the function. Furthermore, RL agents take notoriously more time to train compared to SL agents, and their training in the real world may be challenging, since an agent's interaction with the environment may raise safety and cost complications. **Supervised Learning** offers a high-speed training convergence and does not require specifications on how the agent must reach its goal. An agent's training is based on labeled data, which can be unfavorable for multiple reasons: acquiring and labeling data can be a very slow process (alternatively, one can use predefined data, but the sensor setup has to match the one present in that data); labeled data may contain biases towards certain actions or situations; agents trained with labeled data usually present less adaptability to new environments compared to agents trained with RL.

While with Supervised Learning, ADSs are able to imitate a human driver, with Reinforcement Learning the vehicle is able to learn to drive better than a human, since it is learning by itself. On the other hand, choosing RL can exhaust the resources of a project, without guaranteeing applicability to the real world. **Hybrid learning methods**, like using SL as a pre-training step to a RL agent, try to take advantage of both faster learning processes and better self-adaptation to new environments.

[1]The information from this section was mainly based on the works of Kuutti S. et al. [20] and Grigoresu et al. [19].

2.2.2 Neural Network Architectures

The most common **neural network architectures** used in the field of autonomous driving are **CNN**, **RNN**, **LSTM** and **FCNN**. The exceptionally good results in solving image processing and temporal sequence data processing problems are what make these architectures popular for autonomous vehicle applications. Other [20] feedforward architectures, while less frequent, also appear in the literature. Although RNN and LSTM applications produce good results [21, 22], CNNs are still the broadly used architecture.

CNNs are one of the most successful models for learning complex features from raw input data, especially high dimensional images, showing superior performance on large data sets, as they learn features automatically from training examples, without requiring any manual optimization [23]. Their ability to extract relevant features from images makes them a suitable choice for sensor fusion [24]. However, the autonomous driving task is usually met by problems of temporal dependencies, which is where RNNs are introduced.

RNNs can integrate past information, allowing both visual and temporal dependencies to be learned, therefore solving different autonomous driving scenarios with incomplete information. However, they are not able to memorize long-term dependencies, which is why some works opt for an approach with LSTM networks. Unlike RNNs, LSTMs have a long-term memory that selectively remembers or forgets information which makes them more efficient in dealing with long-range temporal dependencies.

2.2.3 Training algorithms

One of the most commonly used RL algorithms in the ADS context is **Q-learning** or variations, some of the most relevant being **Hierarchical Q-Learning** [25], created to solve problems arising from large state-action spaces, **Double Q-Learning** [26], that solves the poor performance of the Q-learning in stochastic environments, and **SARSA** [13], that prioritizes the safety of the system over optimality. The **Hierarchical Q-Learning** and **SARSA** algorithms were not considered better alternatives for this work's implementation: the state-action space is relatively small and, since the work is exclusively applied in a simulator, optimality is preferred over safety. **Double Q-Learning** was considered a potential improvement to this thesis, since it does not increase the computational complexity. Nonetheless, this improvement was pushed to future work. In the context of autonomous driving, **Deep Q-Networks** (table A.1) are a popular choice, as they make it possible to apply Q-learning methods to a high dimensional state-action space, and optimize the training process with NN as function approximations.

Other popular RL actor-critic algorithms include **Supervised Actor-Critic** (SAC) [27] and **Asynchronous**

Advantage Actor-Critic (A3C) (Table A.1). By having a combination of value-based and policy gradient approaches, these algorithms find a way to have a better performance than the other two algorithms separately. In some works it is also mentioned a subset of Reinforcement Learning called **Imitation Learning** [28]. Some Imitation Learning methods, like **Inverse Reinforcement Learning (IRL)**(table A.1), appear in the ADS context to overcome obstacles like dataset biases and learning complex reward functions.

In Supervised Learning (SL) applications to ADS, **Neural Networks** are the most popular approach. These are used to solve either classification or regression problems [29, 30], or are used as function approximators for the RL algorithm. When using Neural Networks, the most commonly used **optimizers** are **Stochastic Gradient Descent** (SGD), which is the basic algorithm, and its variants, like **AdaBoost**, **AdaGrad** and **Adam**. Although some works claim better results from one or two optimizers, it is still a highly argued subject.

2.2.4 Partial and full vehicle control

In the context of **lateral control**, Supervised Learning with an end-to-end architecture is the most common approach. **Lateral control techniques** are characterized by using the steering angle as output and camera images of the environment as input of the network, generally using multi-layer CNN architectures to process these images. Reinforcement Learning has also been implemented, showing higher adaptability than SL solutions.

One of the most well-known systems for **longitudinal control** is the Adaptive Cruise Control (ACC), and a commonly used solution for these types of systems is a **Deep Neural Networks (DNNs)** combined with RL since it does not require the knowledge of the system's complex model and shows strong adaptability to new complex environments. The most common NN inputs and outputs found in this type of control are relative velocity, acceleration and/or distance as input and desired acceleration or accelerate/break actions as output.

Full vehicle autonomy is a relatively new area of research. Recent research shows that robust high-performing full vehicle autonomy is still out of reach, the main constraint being the significant complexity of the tasks performed by the agent. A variety of learning strategies have been explored, however showing a preference for end-to-end architectures with Supervised Learning and CNNs. The inputs are generally camera images and the outputs are steering and acceleration variables. Ideally, the strategies tested in this context provide robustness to noise, deviations to trajectory and other errors, support to high-speed driving, and allow the system to take more types of inputs into account. Although the results show progress towards these goals, full vehicle autonomy still shows a lack of adequate performance and robustness compared to partial autonomy.

2.3 Challenges

The past sections have shown that, over the last years, there has been a lot of research around the applications of Machine Learning to autonomous vehicle control, which has been supported by tremendous technological growth in many related areas. However, it still faces a significant amount of challenges, especially with real-world applications [20]. This section reviews the biggest **technological challenges** commonly faced by Machine Learning applications in the context of autonomous driving. It is important to note that issues such as artificial intelligent machine ethics, cost efficiency, interaction with humans and lack of legislation and regulation are also present in this research field, but are not the focus of this section.

- **Computation and Dataset:** Two of the biggest obstacles faced by Machine Learning applications are the high computation complexity of the methods and the size of the dataset. Deep Learning methods require large amounts of data and a significant time to train a network adequately. While bigger and more diverse datasets provide more robust and reliable systems, computational complexity grows exponentially with the number of dimensions of an environment (**Curse of Dimensionality**), making the training process slower for more complex environments. Common solutions to reduce training data requirements or training times are based on combining RL with SL: RL provides higher adaptability for the system in new and complex environments while the prior knowledge of SL speeds up the learning process, exploiting the advantages of both methods. It is also important to ensure the **datasets** are diverse and do not include biases. Simulated environments often produce datasets with harmful biases. A solution here is to use methods such as IRL to learn from human driver samples. [19]

- **Architecture and general design:** In Reinforcement Learning applications, defining an adequate reward function is essential but usually very difficult. IRL is known to help with this obstacle since it provides a natural system for optimizing the reward function without the need to define it empirically. However, IRL is usually expensive to run and needs a lot of human driving data to obtain decent results. Also, in most cases, the human driver cannot cover all the possible states that the agent may encounter during training, which makes for an incomplete support [31]. Another challenge faced is reward hacking, which is an effect that occurs when the agent exploits the reward function in a way that does not align with the developer's objectives for the agent [20].

- **Adaptability and generalization:** An autonomous vehicle encounters a very wide range of in-consistent environments, which is why it is vital to ensure the system can generalise well, i.e., interact properly with a completely new environment. **Generalisation** is difficult to guarantee, and it is virtually impossible to test the vehicle in all the scenarios it could encounter. Simulators are usually used to train the system on a diverse set of environments, but these systems often fail to generalise well in real environments [24]. The usual workaround for this problem is to avoid overfitting - when the model fits the training data too well and is not able to generalise well to new data - during training. There are no known methods to choose when to stop training the model (stopping point), but it is advised to implement a three dataset policy: a set for training, a set for validation and a set for testing. By observing the error in the validation dataset, it is possible to infer and avoid overfitting. Validation sets can also help choose between different network architectures and parameters.

- **Verification and Validation:** To properly validate the performance and safety of the system, rigorous testing must be applied. This includes testing in conditions as similar and situations as diverse as the ones encountered by these systems in the real world. Testing in the real world can often be expensive, time-consuming and unsafe. That is why simulation studies are becoming the dominant method of study in this field [20]. However, while cheaper, more flexible and faster, simulators difficult the process of transferring the system to the real world. The developer must ensure the agent's policies do not overfit to the simulation environment, which is why **it is important to validate both the model and the simulation environment**. The quality of the training must also be validated.

- **Safety:** Safety is, undoubtedly, one of the most important properties of an autonomous vehicle. Guaranteeing that these systems are safe is a top priority since a failure or malfunction can have devastating consequences. However, the solutions provided by systems operating with Machine Learning methods are increasingly difficult to interpret and control. The **lack of transparency** of these systems is known as the **black box problem** - if we do not understand how the system makes its decisions, ensuring it does not make unsafe decisions in new environments becomes increasingly difficult [20]. To prepare the vehicle for any dangerous scenario, it is common to train the system in unsafe and unpredictable situations, while always maintaining safety during training and testing in the real world. Approaches to ensure functional safety include combining reinforcement methods, that provides superior adaptability, with traditional control methods, increasing safety through traditional control theory [32].

2.4 Related Work

When designing an intelligent ADS, the most popular approach is a SL-based end-to-end architecture with a complex neural network. Bojarski et. al [33] proposes **an end-to-end architecture with a 9-layered CNN trained with Supervised Learning methods** to map RGB images from a single front-facing camera directly to steering commands for the vehicle. The authors claim the proposed approach to be surprisingly powerful, showing high adaptability to different conditions and environments with a simple architecture and small amounts of training data. However, besides the lack of transparency of an end-to-end architecture, the proposed NN architecture adds great computational complexity to the system, while also requiring large datasets to be created for training.

Although less common, ML methods applied directly to the control module are also present in the literature, including for Proportional Integral Derivative (PID)-controller tuning. Bari S. et. al [34] proposes the use of **a simple NN for online tuning of a PID controller's parameters** in a highly non-linear system such as a Flight Control of a Quadcopter. The proposed NN has three inputs, a hidden layer and the outputs are the gains K_p, K_i and K_d. This work showcases the benefits of applying ML directly in the Control module, and also demonstrates the improvements of ML over conventional offline PID tuning methods. Despite that, the testing environment is simplified and the proposed system's robustness is not comparable to an autonomous driving system.

The works from Panagiotis Kofinas et. al [35] and Shi et. al [36] both propose a RL approach for tuning PID controllers. Shi et. al [36] proposes the **tuning of 2 PID controllers' gains in a simulated cart-pole system, using the Q-Learning algorithm**. The agent is composed of 6 discretized Q-tables (one for each gain of each PID), with an adaptive α method (Delta-Bar-Delta) and ϵ-greedy policy for action selection. This work argues in favour of introducing the adaptive properties of the Q-Learning algorithm directly into the control module through PID tuning. On the other hand, a system with 6 Q-tables requires extensive, and sometimes infeasible training times. Panagiotis Kofinas et. al [35] proposes a **hybrid Zeigler-Nichols (Z-N) Reinforcement Learning-based PID tuner** to control the speed of a DC motor, where the PID gains are set by the Z-N method and then are tuned online through a **fuzzy Q-learning agent**. The agent can increase or decrease the gains from 0% up to 50% of their original values. The agent's states and reward function are defined by the error, and the actions are 3 variables defining the percentage of change of each gain. Fuzzy logic is used to allow the Q-Learning to be applied in a continuous state-action, despite increasing the complexity of the system. The authors argue that using a NN as a function approximator would result in a very slow learning rate and, although simpler, discretizing the action-state space can lead to sudden gain changes, which may cause dangerous behaviours.

The work of this thesis is inspired by [35]: it proposes a RL agent applied in the motion control module to tune the gains of a controller. It is argued that, although it takes more time to train, a RL agent provides desirable adaptability and computational simplicity, while also facilitating the workload by not requiring a dataset to train. Furthermore, discretizing the Q-table and limiting the state-action space reduces the **training times**.

The work of [36] and [35] signify the viability of using a Q-learning algorithm to tune a controller's gains in a simulated environment. Based on that, a **basic Q-learning algorithm with a discretized action-state space** is proposed. As mentioned in [35], this approach enables sudden changes in actions, which could lead to unwanted behaviours or even safety issues in future real-world applications. These issues can be solved by working with a higher-level controller (trajectory controller), instead of the PID controller. In any case, defining the control laws and setting **gain limits** help prevent these behaviours from occurring. Furthermore, applying these ML methods only on the motion control module allows for a high level of control over the system, as other modules may be added to build a ”safety net”, giving the developer even more control over the system.

Despite the advantages, a considerable amount of **hyper-parameters** are involved in the RL agents, especially when training in different environments. This requires extensive trial-and-error testing to achieve the best results, which is why training in a simulation is preferred. On the other hand, delays in the communications between the system and the simulated vehicle lead to a large dispersion of the data retrieved (see 4.1), which causes the training results to be less precise. This compromises the system's performance and even complicates a transition to a physical system.

Chapter 3

System architecture

This chapter presents the proposed system's architecture. This architecture can be divided into 4 modules: the **vehicle simulator**, a **high-level controller**, a **Reinforcement Learning agent** and a **low-level controller**. The simulation consists of a vehicle with multiple sensors attached. The low-level controller drives the vehicle through a predefined reference path (a vector of evenly spaced poses) by imposing values for velocities and steering angle. This controller is responsible for calculating these values, with the goal of optimizing the vehicle's behavior while performing certain maneuvers. The optimization process consists of adjusting the **set of gains** used to calculate the velocities and steering angle values to minimize the average error between the vehicle's pose and the reference path. The RL agent is trained to find the best set of gains for each maneuver. The maneuvers that were tested were **lane changing** (to the right) in a straight road and **circulating in a roundabout**. The high-level controller processes information from the simulator and communicates with the other modules, making a bridge between them.

A diagram of the complete architecture is presented in figure 3.1:

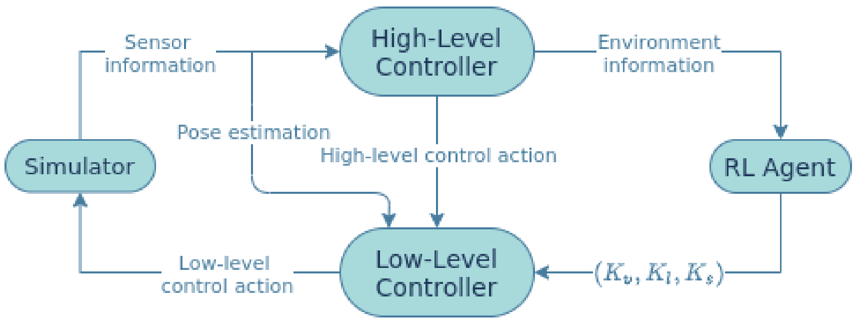


Figure 3.1: Full system architecture

The system works as follows: while the vehicle follows the reference path, the high-level controller collects and processes data from the simulation sensors. Then, if necessary, it sends **high-level control actions** to the low-level controller (see section 3.4). If it is necessary to perform any of the maneuvers, it sends that information to the RL agent, which will then find the appropriate fine-tuned gains for that maneuver. Those **fine-tuned set of gains**, (K_v, K_l, K_s), are then sent to the low-level controller to adjust the **steering angle**, ϕ, the **linear and the angular velocities**, v and w_s - these three values define a **low-level control action**. The simulator also communicates directly with the low-level controller, sending an estimation of the vehicle's current pose. This estimation is made based solely on the vehicle's odometry.

The tuning process is offline, therefore the gains are constant while the vehicle follows the reference path, and will only change if the vehicle is prompted by the High-Level controller to perform one of the mentioned maneuvers.

Essentially, the vehicle is expected to behave as a human driver would when facing different types of environments. The gain adjustments should mimic the adjustment in the driving style a human driver would display.

The next sections go into detail about each of the system's modules.

3.1 Simulator

CARLA is an open-source simulator designed for autonomous driving research. It simulates urban driving environments, with multiple vehicle models, sensors, objects and pedestrians. CARLA is grounded on Unreal Engine [37] to run the simulation. The vehicles inside the simulation have their own controllers - these will be referred to as the **simulation controllers**. The RL agent is trained in the simulated environment, while the low-level controller communicates with the simulation controller to impose velocity and steering angle values.

The simulator has a bridge with ROS. This allows direct communication with the simulated vehicle, through publishers and subscribers from this bridge, but it also provides a way to customize the vehicle setup, which is initialized through a *.launch* file. The proposed setup is a ”Tesla Model 3” vehicle with an obstacle detector sensor located on the vehicle's bumpers, a speedometer, a collision detector sensor and an odometry sensor. Since the obstacle detector sensor is not included in the **CARLA ROS bridge** packages, this sensor is initialized separately from the rest of the setup. Figure 3.2 illustrates the communication system provided by the ROS bridge:

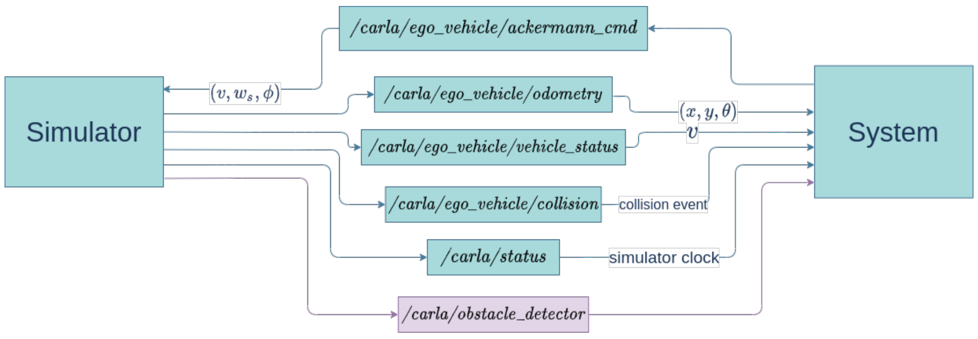

Simulator
System
/carla/ego_vehicle/ackermann_cmd
(v, w_s, ϕ)
/carla/ego_vehicle/odometry
/carla/ego_vehicle/vehicle_status
(x, y, θ)
v
/carla/ego_vehicle/collision
collision event
/carla/status
simulator clock
/carla/obstacle_detector

Figure 3.2: Communication system between the simulator and the system

The system communicates with the simulation controller through **Ackermann Messages**, which consist of 3 main parameters: *steering_angle*, *speed* and *steering_angle_velocity*. Each iteration, the system uses the publisher *"/carla/ego_vehicle/ackermann_cmd"* to send a message with updated values for these parameters. The value for the *steering_angle* is defined as ϕ, for *speed* is v and for *steering_angle_velocity* is ω_s. Then, the simulation controller converts the Ackermann Messages into commands that control acceleration and velocity through an integrated Python based PID.

The simulator communicates back with the client through subscribers, to topics such as *"/carla/ego_vehicle/odometry"* for the vehicle's odometry, *"/carla/ego_vehicle/vehicle_status"* to check the vehicle's current linear velocity, *"/carla/ego_vehicle/collision"* to check for collisions and *"/carla/status"* to keep track of the simulation time. Since the obstacle detector sensor is initialized separately from the rest of the setup, the data is sent to the system by a publisher created in a separate script, which is marked in purple in figure 3.2.

3.1.1 Vehicle model

Figure 3.3 shows the coordinate system of the vehicles inside the CARLA simulator - this will be referred to as the **vehicle frame**. The longitudinal movement of the vehicle is defined by the X-axis, with the positive values corresponding to the forward movement and the negative values to the backward movement. The lateral movement of the vehicle is defined by the Y-axis, with the positive values for movement to the right and negative values for movement to the left. The Z-axis is aligned with the vehicle's center of mass, pointing upwards: Roll, pitch and yaw are defined in the figure by circular arrows. Roll, in blue, is the angle around the X-axis. Pitch, in red, is the angle around the Y-axis. Yaw, in green, is the angle around the Z-axis. Roll and pitch usually stay near zero. Yaw is the angle that defines the vehicle's orientation, with positive values for rotations to the right and negative values for rotations to the left.

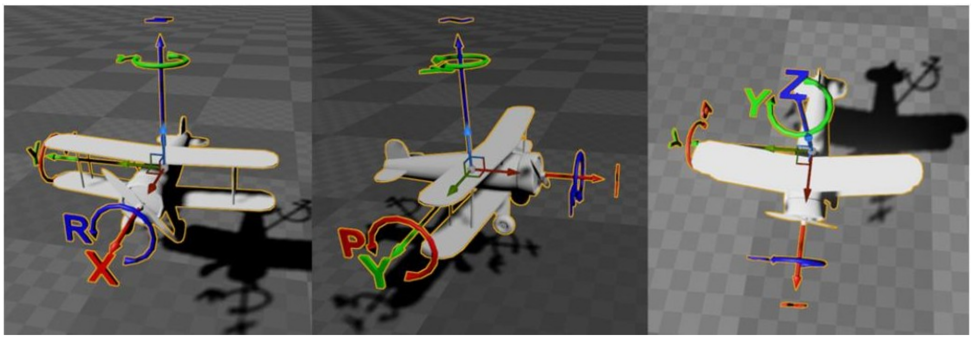


Figure 3.3: CARLA's coordinates system

3.1.2 Simulation time

The simulation has its own clock, conducted by the server. This **simulation time** is generally slower than real time. The time span between two simulation instances, or frames, called the time-step, can be configured to be either variable - running as fast as it can - or fixed. For this project, the simulator runs with a **fixed time-step of 0.01 simulated seconds**, which is equivalent to 100 frames per simulated second. As previously stated, the topic *"/carla/status"* is used to maintain track of the simulation's current frame - if 100 frames have passed since the start of the simulation, 1 simulated second has elapsed.

CARLA is built over a client-server architecture. The server runs the simulation while the client retrieves information and sends commands with changes in the world. The communication between client and server can be asynchronous or synchronous. In this case, these communications run in **synchronous mode**, which means the simulation runs as fast as the client can process the information, and never faster. This allows for synchronization between sensors. The simulator was also configured to wait for a new vehicle control command before processing the next frame (see *synchronous_mode_wait_for_vehicle_control_command* in table B.1). This means that the simulation will always wait for the simulation controller to process the last control command it receives.

Working with the CARLA simulator revealed to be one of the biggest challenges of this work. Choosing the most appropriate configuration was vital to the quality of the work, especially the synchronous mode. By forcing a synchronization between sensors, this mode allowed for more consistent results from the simulator. However, while in this mode, the simulator had to wait for every sensor to process the last information it acquired, which would, in turn, slow the processing speed of the simulator, sometimes making it unfeasible to run longer simulations. To avoid this, any sensor that was not necessary to train the agent was disabled. Even so, initial tests still showed inconsistencies with the controller measurements. The tests performed on the simulator and the low-level controller are presented in section 4.1, evidencing this unwanted **"stochastic behaviour"** and demonstrating how its preponderance decreased by making changes to the code and the simulator configurations. A **guide** to installing and configuring the CARLA simulator can be found in appendix B.

3.2 Low-level controller

The controller module guides the vehicle along a desired trajectory by adjusting the values of the steering angle ϕ, linear velocity, v, and angular velocity, ω_s, in real-time. These adjustments are made using **control laws** that are based on the vehicle model, and the goal is to minimize the error between the reference and the actual pose of the vehicle. The values for v, ω_s and ϕ are then converted into control actions to communicate with the vehicle, through the ROS bridge.

3.2.1 Control laws

The control laws are a function of the error between the reference and the actual pose, in the **vehicle frame** (${}^{b}e$). To find this error, the error in the **world frame** (${}^{w}e$) needs to be calculated first using the following equation:

$$ {}^{w}e = [x_{ref} - x, y_{ref} - y, \theta_{ref} - \theta], \tag{3.1}$$

where $(x_{ref}, y_{ref}, \theta_{ref})$ is the reference pose and (x, y, θ) is the vehicle's current pose. Figure 3.4 illustrates how ${}^{w}e$ is obtained.

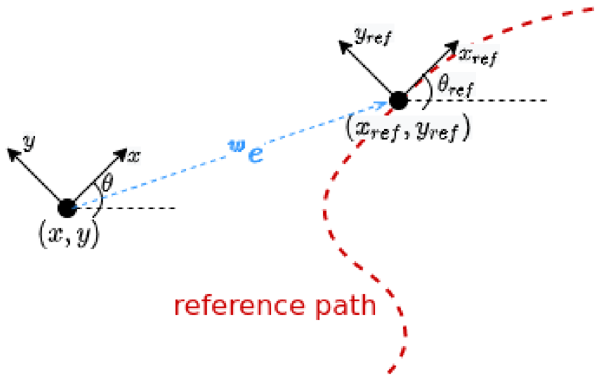


Figure 3.4: Illustration of the error in the world frame, ${}^{w}e$.

To convert the **error in the world frame**, ${}^{w}e$, to the **error in the vehicle frame**, ${}^{b}e$, the following equation is used:

$$ {}^{b}e = R_z(\theta)^{w}e \Leftrightarrow ^{b} e = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} {}^{w}e, \tag{3.2}$$

where $R_z(\theta)$ is a **rotation matrix** that represents a θ degree rotation around the Z axis.

The control laws that will determine the values of v, ω_s and ϕ are defined by the equations:

$$v = K_v {}^b e_x \tag{3.3}$$

$$\omega_s = K_s {}^b e_\theta + K_l {}^b e_y \tag{3.4}$$

$$\phi = \int \omega_s \, dt \tag{3.5}$$

The variables K_v, K_s and K_l are the **trajectory controller gains**. The RL agent will be responsible for tuning these gains, to optimize the trajectory following behaviour of the vehicle. These control laws define, in a broad sense, a proportional controller. This controller does not have an integral or derivative component since they would require more parameters to be tuned, which would increase the training times substantially.

The diagram in figure 3.5 explains how the controller functions. First, the reference pose is obtained from the reference path. The pose chosen as reference is usually not the one immediately closer to the current pose of the vehicle, but one a few meters ahead in the path. The index of the reference pose is calculated according to the following equation:

$$i_r = i_v + ahead, \tag{3.6}$$

where i_r is the index, in the reference path vector, for the reference pose, i_v is the index of the point in the path closest to the actual pose of the vehicle and $ahead$ is the number of indexes from this point to the reference pose. This makes the vehicle continuously "chase" the pose ahead of itself. After defining the reference, the loop begins. It starts by calculating the error in the world frame, ${}^w e$, converting it into the vehicle frame, ${}^b e$, and then using the control laws to calculate v, ω_s and ϕ. It is important to note that the integration, computationally, is obtained by applying the Euler Method [38]. These values are then sent to the simulator, which will process the information (as explained in subsection 3.1) and return the current position and orientation of the vehicle. At any point in the cycle, the High-Level controller can override the linear velocity value to avoid collisions or when the vehicle is breaking the speed limit.

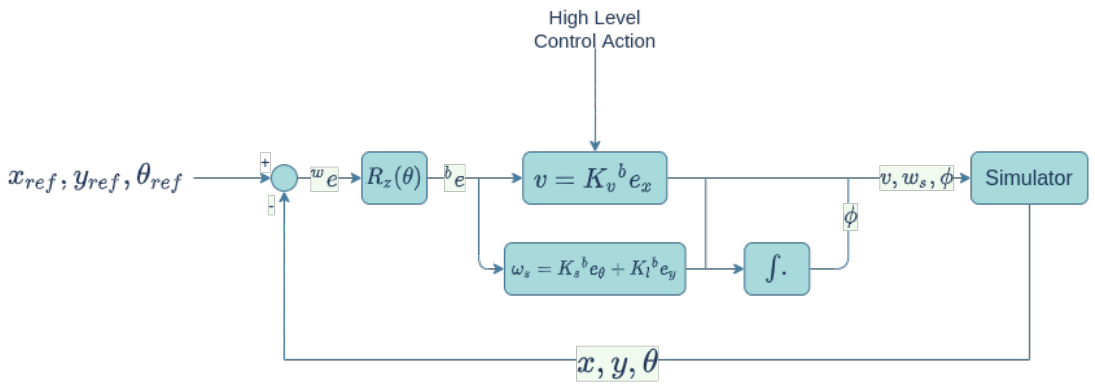


Figure 3.5: Diagram of the controller

The loop repeats until the destination is reached (i.e., if the current position is close enough to the last position in the path), a collision is registered or the simulation time ends. The reference pose is updated each iteration, based on the current position of the vehicle.

3.2.2 Trajectory controller gains

The trajectory controller gains, K_v, K_l and K_s, are the focus of this project. The **performance** of the vehicle when following the trajectory depends heavily on these gains. As a result, the priority will be to find the gains that optimize, or at least improve, this performance when executing the aforementioned maneuvers. In this work, the best gains are considered to be the ones that minimize the average lateral and orientation errors over a period of time.

According to equation (3.3), K_v, **the linear velocity gain**, reflects the impact of the longitudinal error, ${}^b e_x$ on the vehicle's linear velocity, v. This means that the vehicle will try to correct the longitudinal error by increasing its linear velocity, with a proportional component of K_v. Looking now to equation (3.4), **the linear gain**, K_l, is the proportional component of the lateral error and **the steering gain**, K_s, is the proportional component of the orientation error, all weighing on the angular velocity and the steering angle of the vehicle.

3.3 Reinforcement Learning agent

The RL agent is responsible for tuning the trajectory controller gains. This tunning used to be a slow manual task, and would not even ensure the vehicle's most optimized behaviour. The proposed method is automatizing this process using a **simple Q-learning algorithm**, with a **discretized Q-table**, that takes an interval of gains, tests multiple combinations several times and finds the values with which the vehicle presents the best performance. Performance evaluation is translated in the reward function of the algorithm, and will be explained below.

The RL environment is defined as follows:

States

The states are defined by an array with the **average, over the duration of each simulation, of the absolute values of the lateral and orientation errors**:

$$S = [E_y, E_\theta], \tag{3.7}$$

To construct a finite-sized Q-table, these states need to be discretized. For that matter, it is necessary to define a low and a high limit for the values of the errors, as well as how many parts we discretize into. Figure 3.6 illustrates this:

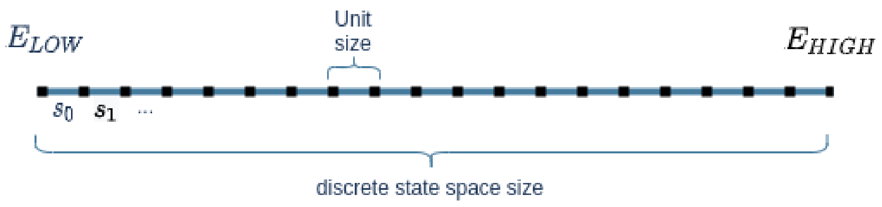


Figure 3.6: Discretization process of the continuous states

where E_{LOW} and E_{HIGH} are arrays with the low and high limits for each of the errors - the range defined for E_y and E_θ is different for the type of maneuver the agent is training. The **discrete state space size** is the number of units this interval is discretized into. For this work, each of the errors is discretized into 40 units. The discrete state, S_K is obtained by the equation:

$$S_k = \frac{[E_y, E_\theta] - E_{LOW}}{unitsize}, \tag{3.8}$$

where $[E_y, E_\theta]$ is the continuous state and $unitsize$ is the size of one discretized unit, which is obtained by the equation:

$$unitsize = \frac{E_{HIGH} - E_{LOW}}{40} \tag{3.9}$$

The average of the longitudinal error, E_x, is not taken into account in the state definition. This is because the performance of the vehicle while performing the two maneuvers mentioned is more significantly influenced by the lateral and orientation errors than by the longitudinal error. Furthermore, a smaller state-action space is more advantageous since it requires less time to train on.

Actions

Each action, $A = [a_0, a_1, a_2]$, is represented by an array that defines either an **increase, decrease or invariance of each of the controller gains**. Since there are 3 gains and 3 possible choices (increase, decrease, maintain), there are 27 different actions. The gains are adjusted by the action array through the following equation:

$$K_v = K_v + h_0 a_0 \tag{3.10}$$

$$K_l = K_l + h_1 a_1 \tag{3.11}$$

$$K_s = K_s + h_2 a_2 \tag{3.12}$$

where a_0, a_1 and a_2 are the action array entries and can take the values -1, 0 or 1. The values h_0, h_1 and h_2 are positive constants that will either be ignored, subtracted or added to the previous value of the gain. The gains can be tuned within a defined range. To reduce computational complexity and training times, this range and the h values were picked so that there were 5 possible values for each gain. This way, the work of this thesis only considers 125 possible combinations of gains.

Terminal condition

In a simple Q-learning algorithm, the terminal condition is usually defined as the current state matching a terminal state. In this context, the most logical terminal state would be $S_0 = [0, 0]$ (both the lateral and orientation average error are null). However, given how difficult it is to achieve this state, a ”less ambitious” criteria must be defined. For that matter, the adopted approach was to consider any state that would come closer to $[0, 0]$ to be the terminal state. As a result, if the current state is closer to S_0 than the closest state recorded so far, then a terminal state was reached.

In order to determine the distance of a state to the state S_0, the algorithm uses a **weighted euclidean distance**:

$$d(S, S_0) = \sqrt{\sum_i w_i (S_i - 0)^2}, \tag{3.13}$$

where w is an array of weights. Since the lateral error values, E_y, are generally 10 times greater than the orientation error values, E_θ, this array of weight has the purpose of balancing the impact of these two errors on the distance. Therefore, the weight array used was $[1, 10]$. Taking these values, equation (3.13) can be rewritten as:

$$d(S, S_0) = \sqrt{|E_y|^2 + 10 * |E_\theta|^2} \tag{3.14}$$

If working as expected, this function guarantees that the algorithm always finds a better (or the best) set of gains in each episode, showing a constant improvement of the performance throughout the episodes. On the other hand, with this terminal condition, it is increasingly more difficult for the algorithm to find a terminal state. At a certain point, it is even impossible to find a new terminal state. For that reason, and to assure feasible training times, if a new terminal state is not found in a certain number of steps, the episode is forced to end.

Reward function

Choosing a good reward function is the most important and most difficult task of designing a Reinforcement Learning agent. The process of designing the function used in this system always followed these criteria:

- Since this function reflects performance evaluation of the vehicle, it should be a function of the average of the error;
- If the average error decreases, then the reward must be positive and if it increases, the reward must be negative;
- The reward should penalize taking a bigger number of actions to reach the terminal state so that the algorithm learns the best gains in the shortest time possible.

After testing several alternatives that met those criteria, the reward function chosen for this work is defined by the equation:

$$R = \frac{1}{1 + d(S', S_0)} - \frac{1}{1 + d(S, S_0)}, \tag{3.15}$$

where $d(S', S_0)$ is the weighted euclidean distance between the new state S' and $[0, 0]$ and $d(S, S_0)$ is the weighted euclidean distance between the previous state S and $[0, 0]$.

This function is based on the one used in Panagiotis Kofinas et. al [35], analyzed in section 2.4. It also takes advantage of the euclidean distance metrics used to define the terminal condition. This function not only meets the criteria pointed previously but also associates bigger or smaller rewards for bigger or smaller reductions in the distance.

Another component is added to the reward function to account for collisions: if a collision is registered, the reward is decreased by a defined value.

3.3.1 Training method

The RL agent was trained to perform two different maneuvers: a lane changing maneuver in a straight road and driving in a roundabout. Figures 3.7 and 3.8 show the **reference path** used to train each of these maneuvers, in blue, as well as the initial position of the vehicle, in red. It is important to note that, in both settings, the vehicle starts with an orientation error of zero. Other initial orientation errors are not explored in this work.

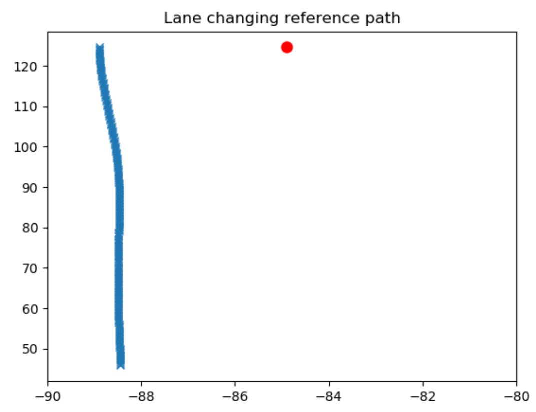


Figure 3.7: Reference path for the lane changing maneuver

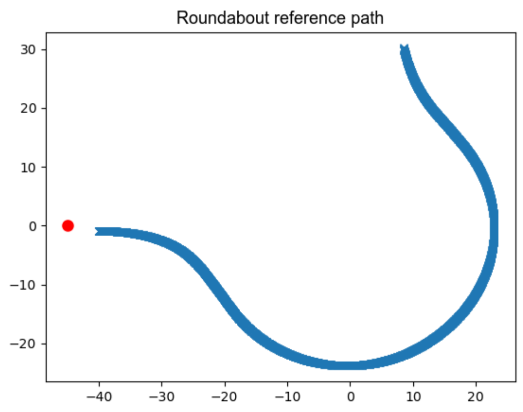


Figure 3.8: Reference path used for driving in a roundabout

As shown in figure 3.7 , the vehicle's initial position in the first environment is 4 meters to the left of the reference path. To correct this 4-meter difference, the controller forces the vehicle to change lanes.In the second environment, in figure 3.8, the vehicle's initial pose is 0.9 meters to the left and 5 meters behind the reference path. Assuming appropriately tuned gains, the vehicle is expected, on the first setting, to return to the right lane as fast as possible, with minimal oscillation, and, on the second setting, to navigate the roundabout properly without colliding with the pavement. The initial positions are set a few meters away from the reference path to force the vehicle to correct its position immediately after departure. The ability to do that properly is important to evaluate the quality of the set of gains - lower gain values will take a longer time to correct this error, increasing the average error and decreasing the associated reward.

The **algorithm used to train** the RL agent is shown by the diagram in figure 3.9:

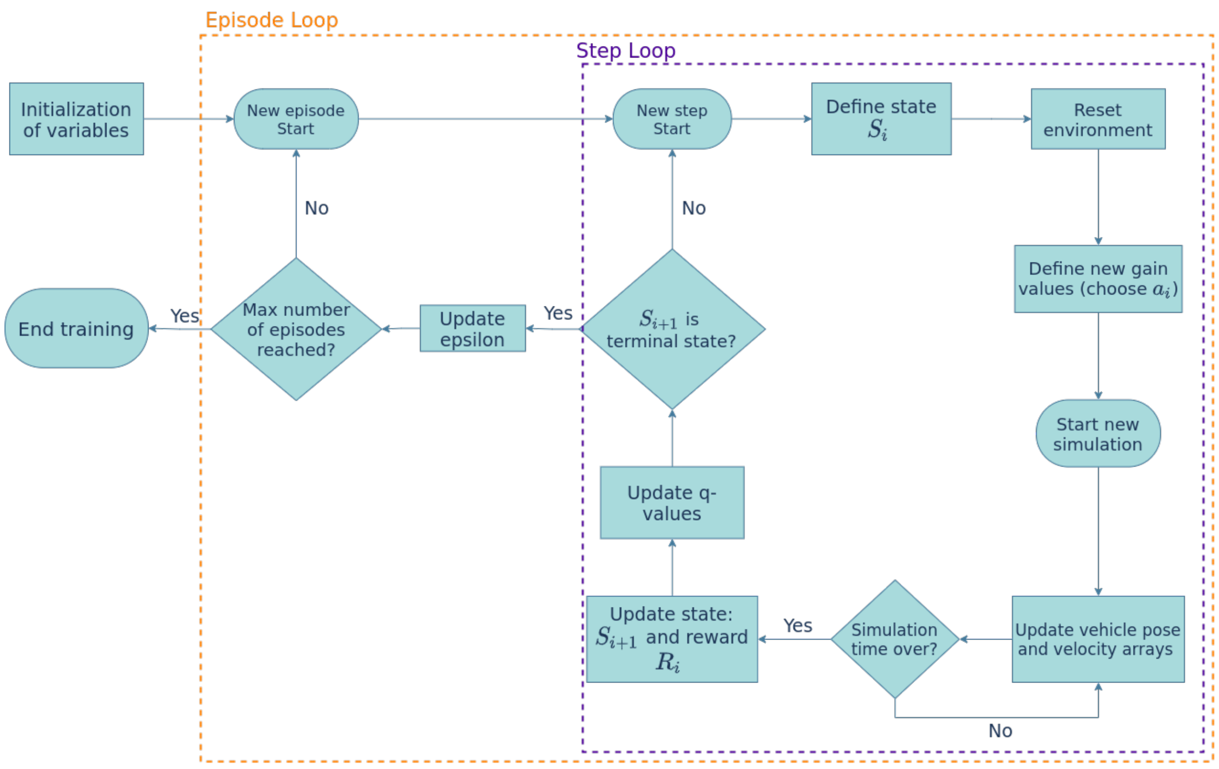

Episode Loop
Step Loop
Initialization of variables
New episode Start
New step Start
Define state S_i
Reset environment
Define new gain values (choose a_i)
Start new simulation
Update vehicle pose and velocity arrays
Simulation time over?
Yes
No
Update state: S_{i+1} and reward R_i
Update q-values
S_{i+1} is terminal state?
No
Yes
Update epsilon
Max number of episodes reached?
No
Yes
End training

Figure 3.9: Diagram of the training algorithm

The agent was trained over a certain number of **episodes**, each of which is divided by **steps**. Each step starts by defining a current state, S_i, based on the last average of errors and re-spawning the vehicle in the initial position and orientation. Then, an action is taken and the new gains are defined. After that, with the new set of gains, a new simulation starts, with the system's controller guiding the vehicle, correcting the error to the reference path. The simulation runs for a period of time (see *loop time* in table 4.1), after which it stops and the new state, S_{i+1}, based on the new average of the errors, and the reward, R_i, are updated. With these values, the Q-table is updated as explained in section 1.3. If the new state, S_{i+1} does not satisfy the terminal condition, then this cycle repeats in a new step. Otherwise, the episode ends.

Section 4.2 discusses the results of the training.

3.4 High-level controller

The high-level controller works as both an **event manager** and a **safety module**, making a bridge between the simulator and other modules and preventing unwanted behaviours from occurring. It performs **two functions**:

- Determine if the vehicle needs to perform any maneuver that the Reinforcement Learning agent has been trained to respond to, based on the map zone the vehicle is currently in;

- Enforce high-level control actions on the low-level controller, preventing the vehicle from performing undesirable and unsafe behaviors. These actions include enforcing a speed limit, controlling the steering angle range and avoiding frontal collisions by overriding the low-level control actions in specific situations.

The controller takes as input the data from the speedometer, obstacle detector and odometry sensors. It processes this information and communicates with the RL agent and the Low-Level controller. The diagram on 3.10 demonstrates how this controller works:

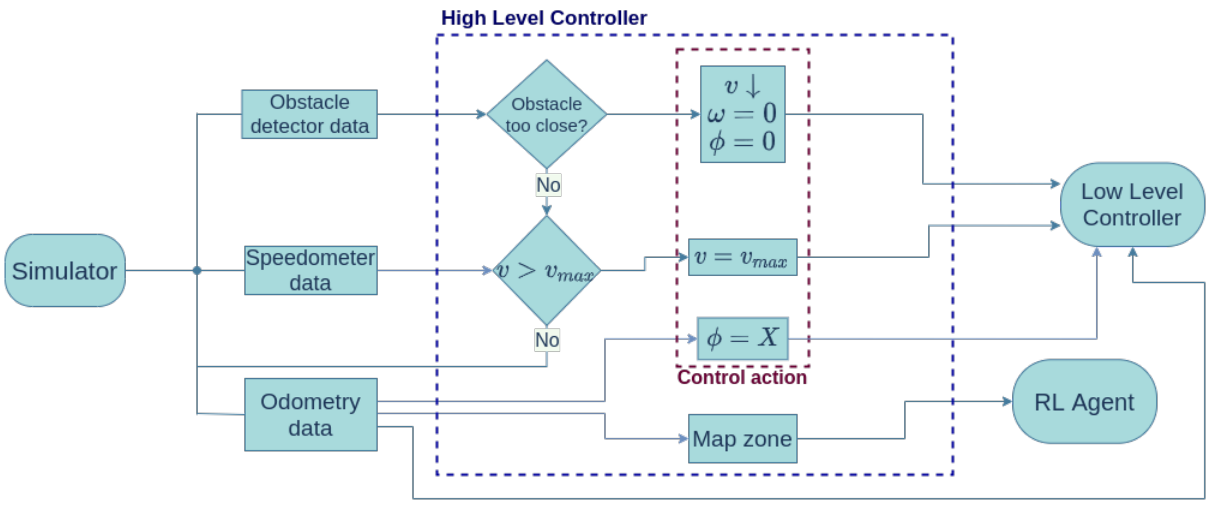


Figure 3.10: Diagram of the high-level controller's internal operations

To test the controller's functions, a portion of the map was divided into **zones**, each of which associated with an event - performing a maneuver, avoiding a frontal collision, or controlling the steering angle. Velocity limits are imposed on the whole map. Figure 3.11 shows the organization of the map:

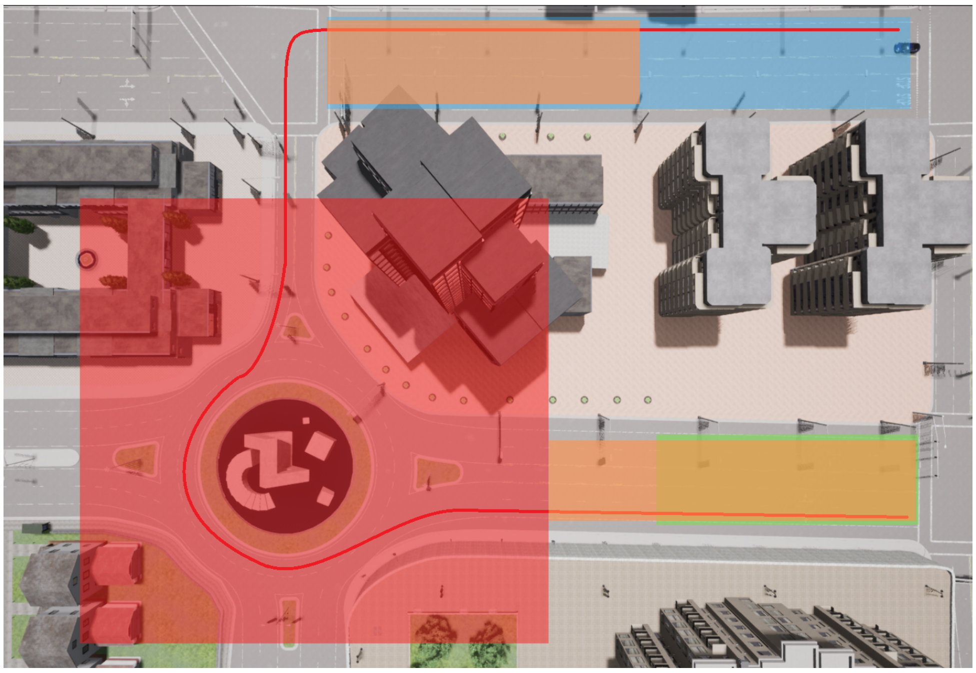


Figure 3.11: Simulation map division

The proposed environment has the goal of testing each of the high-level controller functions sequentially. The map is divided into 4 types of zones: the **blue zone**, where the vehicle performs a lane change, the **red zone**, where the vehicle navigates in a roundabout, the **green zone**, where the collision avoidance function is tested and the **orange zones**, where the steering angle range limit is temporarily reduced. The reference path is marked by a red line. If the controller is working as intended, the gains should adjust to the appropriate maneuver when entering the blue and red zone. When entering the green zone, the vehicle should detect a vehicle in front of it and safely stop. When inside any of the orange zones, the steering angle limit is reduced, going back to its original value when the vehicle is outside those areas.

3.4.1 Collision avoidance

The method used to **avoid frontal collisions** works with the **obstacle detector sensor**, incorporated into the simulation vehicle. The method of detection used is called "Sphere Trace by Channel", present in the Unreal Engine's libraries [39]. Figures 3.12 and 3.13 illustrate this method:

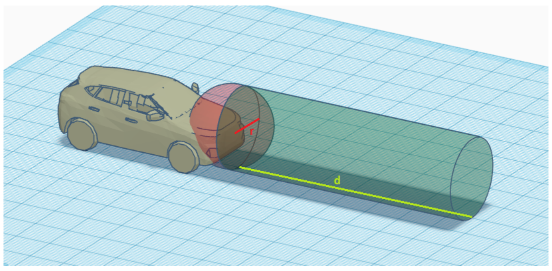


Figure 3.12: 3D view of the obstacle detector sensor using the sphere trace method

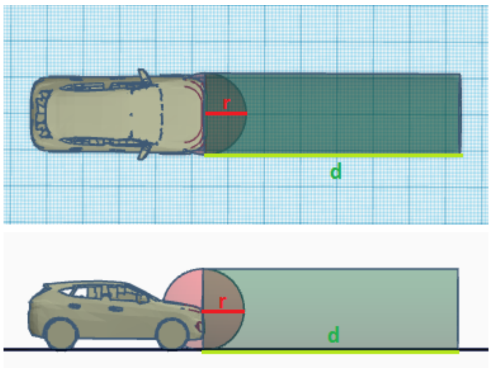


Figure 3.13: Top and side views of the obstacle detector sensor

The obstacle detector sensor has two attributes: *distance* and *hit_radius*, which are represented in figures 3.12 and 3.13 in green by a "d" and in red by an "r", respectively. Any object that is inside the sphere or cylinder will trigger the sensor, which will output the type of object and the distance from the sensor to that object. If any part of the obstacle enters the sphere, the sensor outputs a distance equal to zero. If the distance is less than a defined threshold, the High-Level controller will overwrite the Low-level control actions and lower the linear velocity until the vehicle stops. It will also impose $\omega = 0$ and $\phi = 0$. The zone used to test this function is defined as:

$$\text{Green Zone} \Leftrightarrow -7 < x < 10 \;\;\&\;\; 74.0 < y < 127.4 \tag{3.16}$$

3.4.2 Maneuver identification

Since it is outside the scope of this project, the **maneuver identification method** is very simplified, with the sole purpose of testing the interaction between the simulator and the RL agent. It is programmed to work specifically inside the environment represented in 3.11. The controller uses the vehicle's position to identify if it is inside the blue or the red zone. The zones are defined by the following constraints:

$$\text{Red Zone} \Leftrightarrow -58 < x < 33 \;\;\&\;\; -39 < y < 50 \tag{3.17}$$

$$\text{Blue Zone} \Leftrightarrow -90.5 < x < -73 \;\;\&\;\; 8.5 < y < 127.4 \tag{3.18}$$

The controller communicates with the agent through a set of flags. If the vehicle is inside one of these zones, the controller triggers the corresponding flag, which makes the RL agent adjust the gains to the appropriate ones.

3.4.3 Steering angle range limit

The high-level controller is also programmed to **change the limits of the steering angle range**, depending on the type of zone. In straight lanes, where the vehicle is driving without obstacles in the way, the steering angle range is reduced to minimize the vehicle's oscillation. On the other hand, during maneuvers, this range must be wide enough to allow for faster orientation changes. The zones marked in orange are zones where this range limit is reduced, and are defined as follows:

$$\text{Orange Zone1} \Leftrightarrow -90.5 < x < -73 \;\;\&\;\; 8.5 < y < 73.0 \tag{3.19}$$

$$\text{Orange Zone2} \Leftrightarrow -7 < x < 10 \;\;\&\;\; 50 < y < 127.4 \tag{3.20}$$

Chapter 4

Results and Discussion

This chapter discusses the tests performed to evaluate each of the system's modules. The simulator was tested while working with the system's controller (section 4.1). The RL agent was trained while performing two maneuvers (section 4.2). Finally, the full system is tested (section 4.3) to assess if the high-level controller functions are working properly (obstacles are detected, gain values are changed appropriately and maximum velocity and steering angle are managed).

4.1 CARLA simulator assessment

A series of tests were carried out to assess the simulator's performance while working with the proposed controller. The experiments were designed to evaluate the simulator's odometry precision and were initially performed in the environment of figure 3.7. With that goal, a code with the following structure was used:

1. Define a set of fixed gain values;
2. Spawn the vehicle in an initial position and orientation;
3. Run the control loop (see figure 3.5) to follow a predefined reference path;
4. At each iteration of the loop, register the absolute value of the lateral error, e_y;
5. When the error in the y axis of the vehicle frame (${}^{b}e_y$) is less than 0.1m, end the control loop;
6. At the end of each episode, register the average of the lateral error, E_y;
7. Repeat this for several episodes, for the same set of gains, initial pose, and reference path.

In ideal conditions, the vehicle is expected to perform exactly the same way in each episode. If the trajectory, the registered errors and the time of each cycle is identical, it means the simulator has perfect precision. However, many factors, such as server-client latency, lower this precision. The initial tests presented the following results:

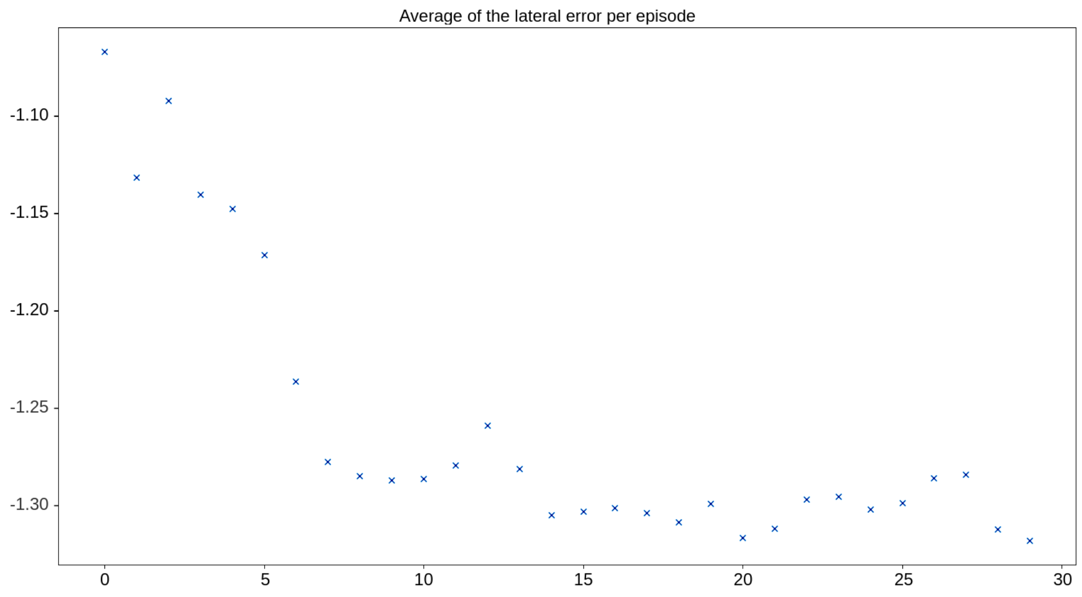


Figure 4.1: Error per episode of the initial tests

Figure 4.1 shows the average error of the vehicle in the y axis, E_y, for each episode. It is possible to see a peek of values in the first episodes, and then a cluster of values between -1.25 and -1.31 in the following episodes. The values spread unevenly between approximately -1.1 and -1.32, an interval of 0.22 meters. The standard deviation of these values is $\sigma = 0.072$. High deviation values can compromise the Q-learning agent's quality.

After some investigation, the high deviation was attributed to a lack of code optimization and delay in communications between the client and the simulator. Different simulator settings were tested and adjustments were made to the code, showing improvements to the processing times and computational complexity. Furthermore, some of the system's parameters showed to have some influence on this dispersion as well, particularly those related to path following. The values of these parameters were picked to decrease the deviation values as much as possible. The reference paths were designed so that each of the points was equally distanced from each other, resulting in a more fluid path following behaviour and therefore less inconsistencies.

The consequence of all these changes was a new set of tests that showed a more satisfactory result:

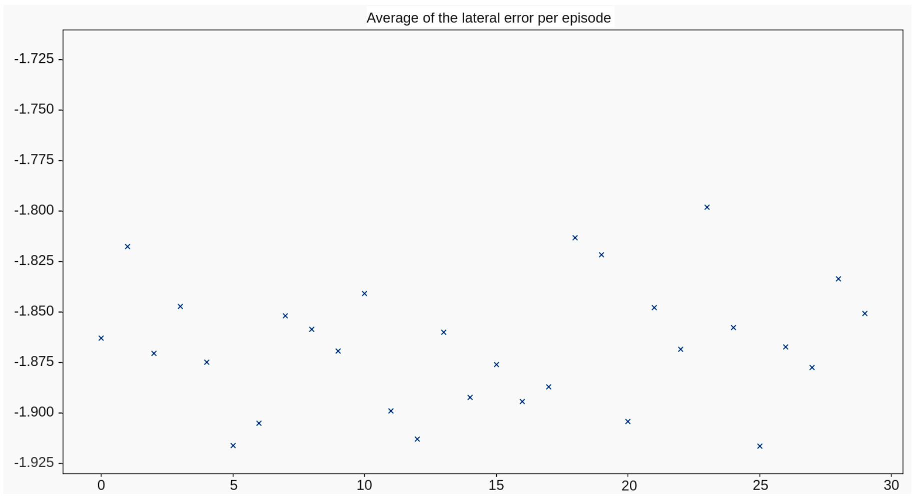


Figure 4.2: Error per episode of the final tests

Figure 4.2 shows the error values per episode of one of the last tests, after the modifications. A size of 0.22 units was imposed on the y axis, so as to match the perspective of figure 4.1. There is a dispersion between -1.8 and -1.91, approximately, a difference of 0.11 meters. Although the cluster on the first test was less disperse than the values in this test, there were no peeks registered. The result was a significant improvement, with a deviation of $\sigma = 0.031$, less than half of the previous tests[1].

More tests were made to the controller's performance during the training of the RL agent, to assess if the previous deviation values were not altered. While the agent was training in the first environment (fig. 3.7), a set of gains were selected to make this assessment:

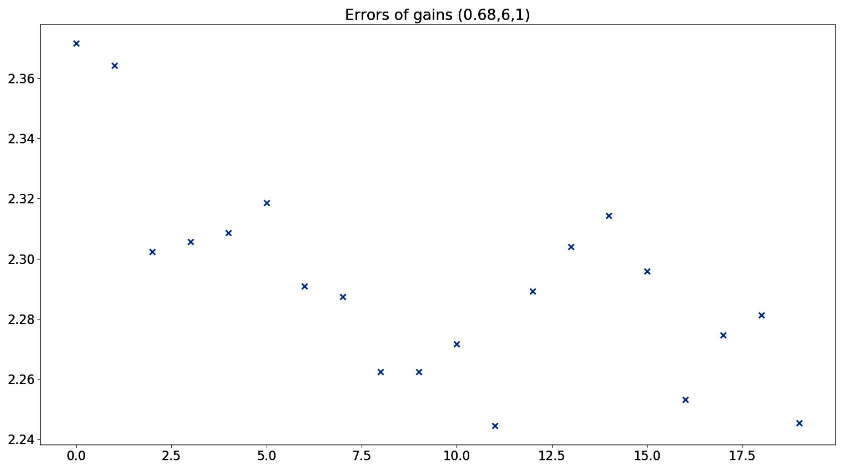


Figure 4.3: Errors of the gains (0.68,6,1) during training in first environment

[1] Although these two tests were made with the same set of gains, vehicle and reference path, the initial position of the first test is 3 meters away from the reference, while in the second test it was 4 meters away. This explains the difference in the average error values.

Figure 4.3 shows all the average lateral errors (E_y) registered by the set of gains (0.68,6,1). The values are distributed between the errors 2.24 and 2.37, a difference of 0.13 meters, and the deviation is approximately $\sigma = 0.034$. These results conclude that the deviation was also low during the training process, therefore this "stochastic behaviour" is expected to have little impact on the agent's training quality.

Similar tests were made during the training of the agent in the second environment (fig. 3.8). These tests were used to tune some of the parameters to the new environment. The results are shown below:

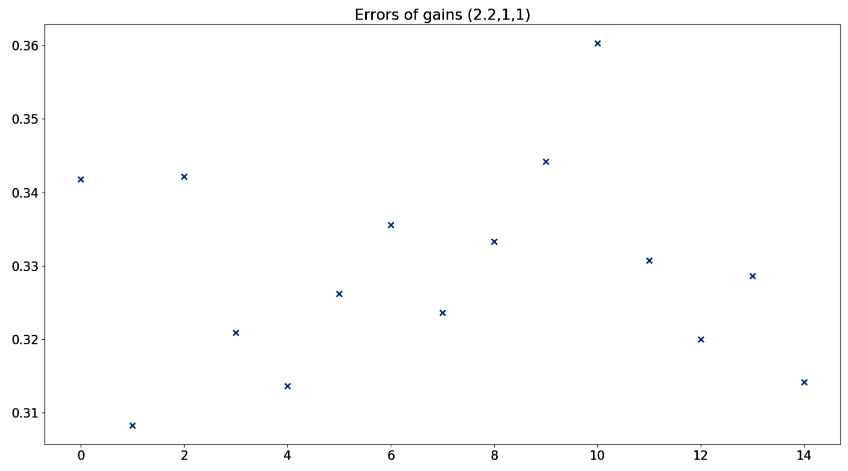


Figure 4.4: Error of the gains (2.2,1,1) during training in the second environment

Figure 4.4 shows all the average lateral errors (E_y) registered by the set of gains (2.2,1,1). The values are distributed between the errors 0.31 and 0.36, a difference of 0.05 meters, and the deviation is approximately $\sigma = 0.0138$. With these results, it is possible to conclude that the "stochastic behaviour" will have an even smaller impact on the agent's training in the second environment.

4.2 RL agent training

The training algorithm involves several parameters, either associated with the Q-learning algorithm or with the controller. As a result, the agent was trained multiple times to test different values, in a process of trial-and-error. To facilitate this process, information relative to each set of gains was registered at the end of each training. This information includes, for each set of gains, the average of the errors E_y and E_θ, the average vehicle velocity and the number of times that set was tested. Based on this, the values chosen for the parameters, for each of the environments of figures 3.7 and 3.8, are shown in table 4.1 :

	Description	Lane Change	Roundabout
Loop time (sim. secs)	Duration of a simulation in each step	5	30
α	Learning Rate (sec.1.3)	$\frac{1}{(n+1)^{0.6}}$	$\frac{1}{(n+1)^{0.6}}$
γ	Discount Factor (sec.1.3)	0.9	0.9
E_{LOW} **(m)**	Minimum state values (sec.3.3)	[0 , 0]	[0 , 0]
E_{HIGH} **(m)**	Maximum state values (sec.3.3)	[4 , 0.4]	[1 , 0.1]
K_{min}	$[K_{v_{min}}, K_{l_{min}}, K_{s_{min}}]$	[0.1, 1, 1]	[1, 1, 1]
K_{max}	$[K_{v_{max}}, K_{l_{max}}, K_{s_{max}}]$	[2.42, 21, 21]	[5.8, 21, 21]
$[h_0, h_1, h_2]$	Positive constants that define the action values (eq. (3.10)-(3.12))	[0.58, 5, 5]	[1.2, 5, 5]
default ϕ range (°)	Default value for the steering angle range	30	30
ϵ (start)	Starting value of ϵ	1	1
ϵ **decay**	Decay value of ϵ per episode	$\frac{1}{n/2}$	$\frac{1}{n/2}$
Step Limit	Limit of steps per episode	120	100

Table 4.1: Table of parameter values for each of the training environments,where n is the number of episodes

The ***Loop time*** values were chosen to be as low as possible and shorten the training times, but high enough so that the full maneuver can be executed and the vehicle's performance properly evaluated. The reference values for α and γ found in the literature are usually constant, generally around 0.1 and 0.9, respectively, but to respect the convergence conditions (sec.section 1.3), a polynomial learning late is used - the polynomial function with an exponent of 0.6 was chosen based on the results from [40] for a low number of steps and $\gamma = 0.9$. The ϵ decay speed is chosen to balance between longer training times and incomplete learning processes. The $K_{min,max}$ values were set by testing the controller separately from the agent and choosing the range of values that better suited each maneuver. The h constants were defined so that each gain had 5 discrete values. The $E_{HIGH,LOW}$ values were set with the goal of using all the state space during training, therefore adjusting the error range to the one found during each training. The ϕ values were empirically chosen to match a realistic range in a vehicle. The ϵ has a starting value, ϵ **start**, which decreases each episode by a value of ϵ **decay**. Both of these parameters' values were also chosen empirically.

As mentioned in Chapter 3, the algorithm considers a new state terminal when its distance to the state $S_0 = (0, 0)$ is lower than the lowest one registered so far. However, early training showed that finding states closer to S_0 becomes a near-impossible task after a few episodes. For that matter, the terminal state condition considers a window of values close enough to the lowest distance registered. So, if the last terminal state registered $dist_{min}$ as the lowest distance to S_0, then the terminal state condition would be defined as:

$$\text{if } dist' \leq dist_{min} + WIN \text{ then } S' \longrightarrow S_t \tag{4.1}$$

where $dist'$ is the distance between the new state, S' and $(0, 0)$ and S_t represents a terminal state. WIN

is the range of the window of values mentioned before, which was adjusted to balance the number of steps it would take each episode to find a new terminal state. If many episodes reached the step limit without finding a new terminal state, then this window was slowly increased until that occurrence is rare.

4.2.1 Lane Changing Training

Using the algorithm explained in 3.3.1, the agent is trained to find the set of gains that minimize the error while the vehicle changes to the lane on the right, following the reference path illustrated in figure 3.7. After tuning the values of the parameters, the agent was trained for 30 episodes, with a window size $WIN = 0.01$. The training time of this test was 18 hours and 30 minutes, in total. It performed 1225 tests to 109 different sets of gains, about 87% of all the gain combinations. The range of gains, $K_{min,max}$ and the h constants define the following values for each gain:

$$K_v \in [0.1, 0.68, 1.26, 1.84, 2.42] \tag{4.2}$$

$$K_{l,s} \in [1, 6, 11, 16, 21] \tag{4.3}$$

Figure 4.5 shows the sum of rewards of each episode of the training, also known as the learning curve:

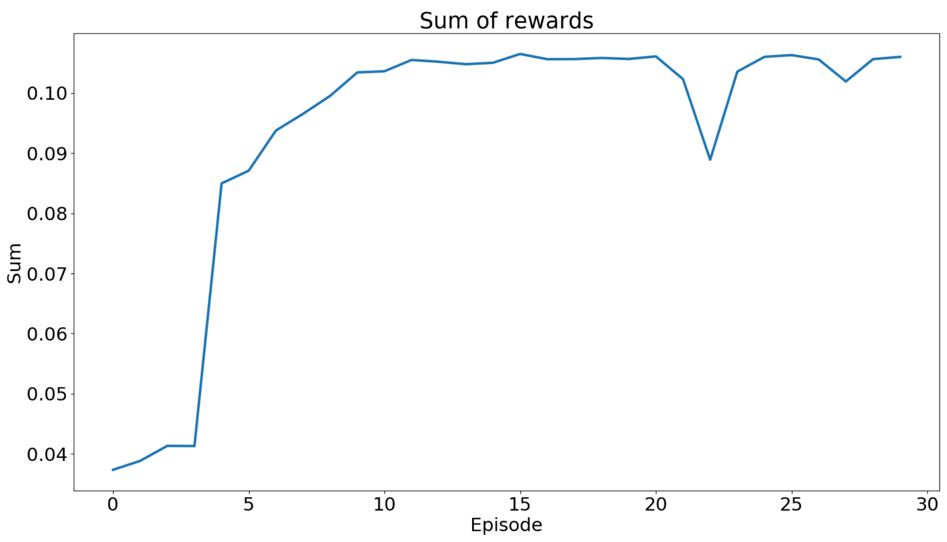


Figure 4.5: Lane changing training: Sum of all the rewards of each episode

As explained in [36], the sum of rewards is an indicator of the efficiency of the algorithm applied. If the sum of rewards of an episode is higher than the previous one it means that the sequence of actions taken was, on average, better than the previous episode. This suggests that the agent is learning and improving its ability to choose better actions. On the other hand, if the sum of rewards is lower on the next episode the sequence of actions taken was averagely worse than the previous episode. There is also the case where the agent performs an action that produces a very high or very low reward, greatly increasing or decreasing the sum of rewards, giving the impression that the sequence of actions is better or worse than it is. However, it is argued that using the reward function in (3.15) with small increments of the gain values prevents these situations from happening.

Figure 4.5 shows a convergence of the learning curve, which implies the success of the algorithm. The graph shows a substantial increase in the sum of rewards for the first few episodes and convergence after that, around the value 0.105, reaching its maximum value, 0.1065, in episode 15. It is believed that this rapid convergence is due to the small dimension of the action-state space, as well as the function used to deduce the learning rate per episode (table 4.1), which was designed to speed up the learning process. Episodes 21, 23 and 27 show a slight drop in value, and episode 22 shows a substantial drop. Unlike the rest of the episodes, in these three episodes, the agent reaches the step limit of 120 without finding a terminal state, which explains the drops.

Assuming the agent is working as expected, the gains chosen after convergence are the ones that produce the lowest errors, i.e., the best gains to perform the specific maneuver. Figure 4.6 shows the gains that produced each terminal state - these will be referred to as **terminal gains**:

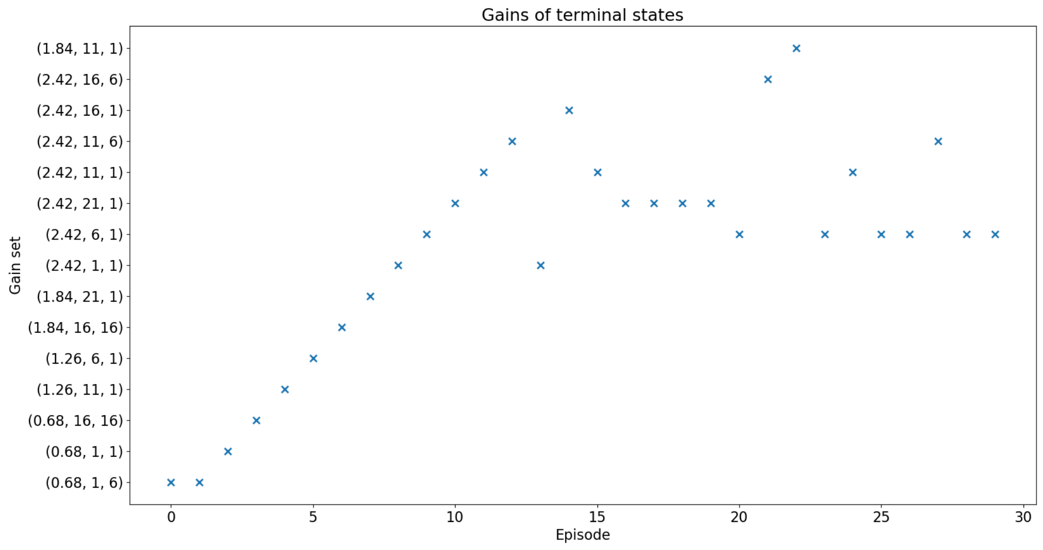


Figure 4.6: Lane changing training: gains of all the terminal states

The agent starts by reaching the terminal state with less efficient sets of gains, progressively improving those gains, showing a preference for the gains $(2.42, 6, 1)$ and $(2.42, 21, 1)$ after converging.

The center of mass[2] of the cluster of gains is chosen to define the set of gains used to test the system in the blue zone. Only the gains after the learning phase are considered, i.e., all terminal gains from episode 15 onward, removing the episodes where a terminal state is not reached (episodes 21, 22, 23 and 27). The resulting set of gains is $\mathbf{(2.42, 11.67, 1.33)}$. Before testing the system, the chosen set of gains is validated separately. The validation process consists of evaluating the performance of this set of gains while performing the maneuver and comparing that with the performance of other sets of gains.

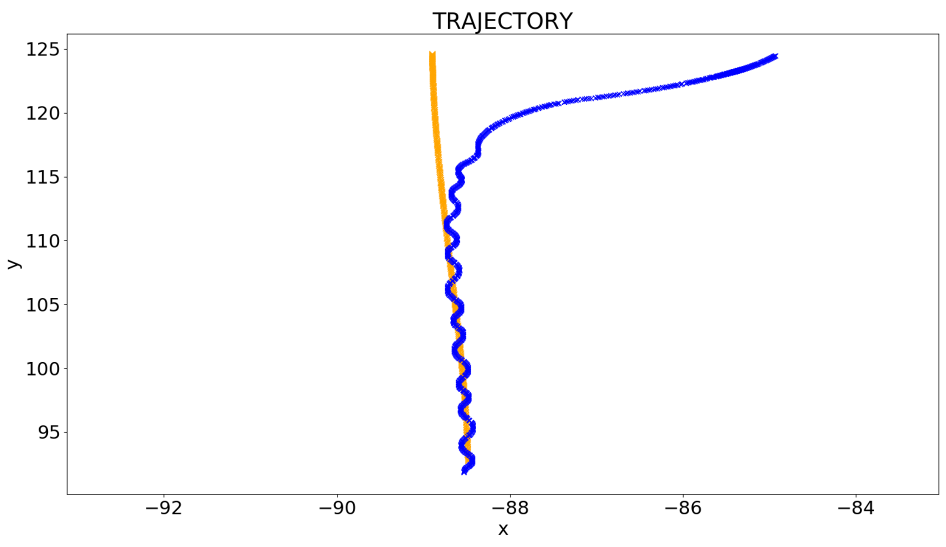
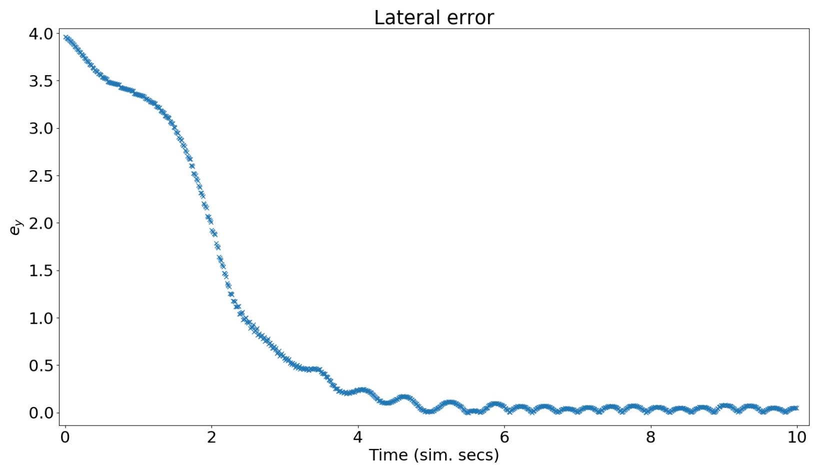


Figure 4.7: Lane changing Validation Test: Trajectory

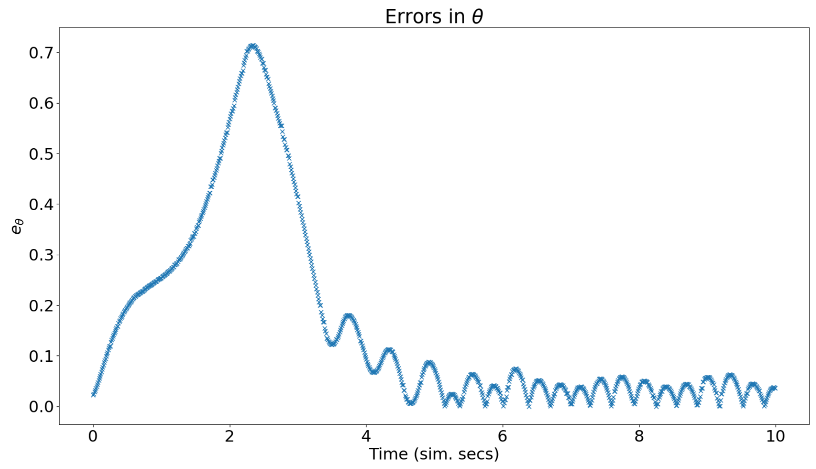
The first test consisted of performing the lane changing maneuver while using the chosen set of gains, for 10 simulated seconds. The vehicle starts in position $(-84.9, 124.6)$, with the correct orientation. Figure 4.7 shows the trajectory performed by the vehicle in blue and the reference path in orange. The figure suggests that the system can perform the maneuver efficiently, completing the lane changing approximately 10 meters ahead of the initial position. The trajectory also shows some oscillation after the lane changing, fixed by controlling the steering angle range, which is implemented and tested later in this chapter.

The figures below show the absolute values of the lateral and orientation errors registered in this test:

[2]This method was chosen to take into account all of the good sets of gains evenly. Given the nature of the controller, it is not expected that the chosen "middle" gains produce substantially different behaviour. Additionally, the defined range of gain values guarantees that these gains will not result in dangerous behaviour. Alternatively, one can choose the most popular set of gains.



(a) Absolute value of lateral error, e_y

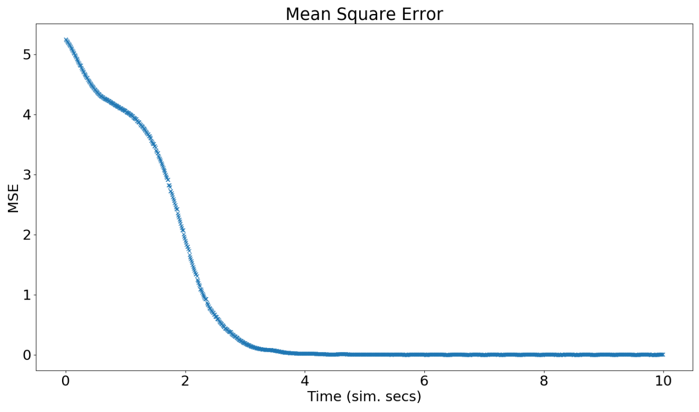


(b) Absolute value of orientation error, e_θ

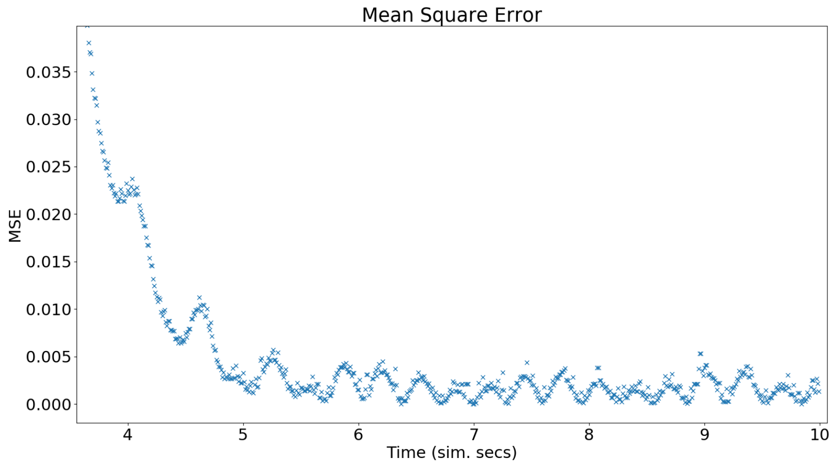
Figure 4.8: Lane Changing Validation Test: Lateral and Orientation errors

Figure 4.8(a) shows that the system was able to correct the 4-meter lateral error in about 4 simulated seconds with the chosen set of gains. After the initial error is corrected, the error stabilizes below 0.1 meters. Figure 4.8(b) shows an orientation error spiking at 0.7rad ($\simeq 40.1$ degrees), during the lane changing maneuver, and then stabilizing at less than 0.1 rad($\simeq 5.73$ degrees).

Figure 4.9 shows the Mean Square Error (MSE), calculated as $MSE = \frac{{}^be_x{}^2 + {}^be_y{}^2 + {}^be_\theta{}^2}{3}$, registered during the test:



(a) MSE values



(b) MSE values after the initial error (zoomed)

Figure 4.9: Lane Changing Validation Test: Mean Square Error values

In conformity with the error values of figures 4.8, the MSE starts at around 5.2 and rapidly drops. After the initial error, it stabilizes at less than 0.006, as shown in figure 4.9(b).

Table 4.2 presents the average MSE of some sets of gains, including the chosen one, in bold. As mentioned before, some dispersion is observed in the error values produced by the simulator. For that matter, the table presents the highest average MSE registered out of 10 samples, excluding the first second of the simulation:

	Average MSE	σ
$(0.68, 1, 1)$	2.723	0.407
$(0.68, 21, 21)$	2.194	0.327
$(1.26, 11, 11)$	1.294	0.174
$(1.84, 16, 16)$	0.8420	0.135
$(1.84, 11, 1)$	0.7809	0.1203
$\mathbf{(2.42, 11.67, 1.33)}$	$\mathbf{0.4658}$	$\mathbf{0.0328}$
$(2.42, 21, 21)$	0.5628	0.05

Table 4.2: Average MSE of set of gain from the lane changing validation tests

Table 4.2 compares the chosen gains with other sets of gains spread through the range of values. The third column presents the standard deviation values, σ, from the 10 samples gathered from each set of gains. A qualitative analysis of these values reveal that the chosen set presents the lowest average MSE, with gains $(2.42, 21, 21)$ following close behind. Also, the closer to the chosen gains, the lower the σ values are. The table also suggests that differences in the value of K_v have more impact on the results, with the biggest decreases observed between $K_v = 0.68$ and $K_v = 1.26$.

Overall, the performance of the system working with the chosen gains shows low lateral errors and MSE values. Table 4.2 suggests that the agent's choice of gains is around the values that minimize the average MSE. Although the deviation observed in the error values lowers the accuracy of the agent, the lowest MSE scores frequently belong to the chosen set of gains.

4.2.2 Roundabout Navigation Training

After the last environment, the agent is trained to find the set of gains that minimize the error while the vehicle navigates a roundabout, following the reference path illustrated in figure 3.8. The values of the parameters had to be adapted to the new environment, as demonstrated in table 4.1. The agent was trained for 20 episodes, with a window size $WIN = 0.007$. The range of gains, $K_{min,max}$ and the h constants define the following values for each gain:

$$K_v \in [1, 2.2, 3.4, 4.6, 5.8] \tag{4.4}$$

$$K_{l,s} \in [1, 6, 11, 16, 21] \tag{4.5}$$

The training time of this test was approximately 16 hours. 317 steps were performed to 94 different sets of gains, out of 125 (75%). Figure 4.10 shows the sum of rewards of each episode of the training:

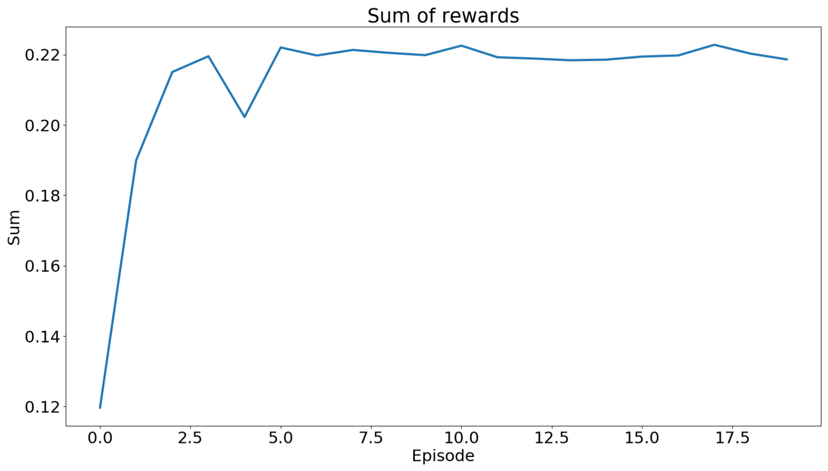


Figure 4.10: Roundabout training: Sum of all the rewards of each episode

Similar to the last training, the graphs show a substantial increase in the sum of rewards for the first few episodes, converging at around 0.22 of sum of reward, its maximum value being 0.2228, in episode 17. Once again, this learning curve suggests the agent is learning as intended, and converging to a set of gains in just about 10 episodes, except for episode 4, which shows a drop, once again explained by the step limit being reached.

Figure 4.11 shows the gains that produced the terminal state of each episode:

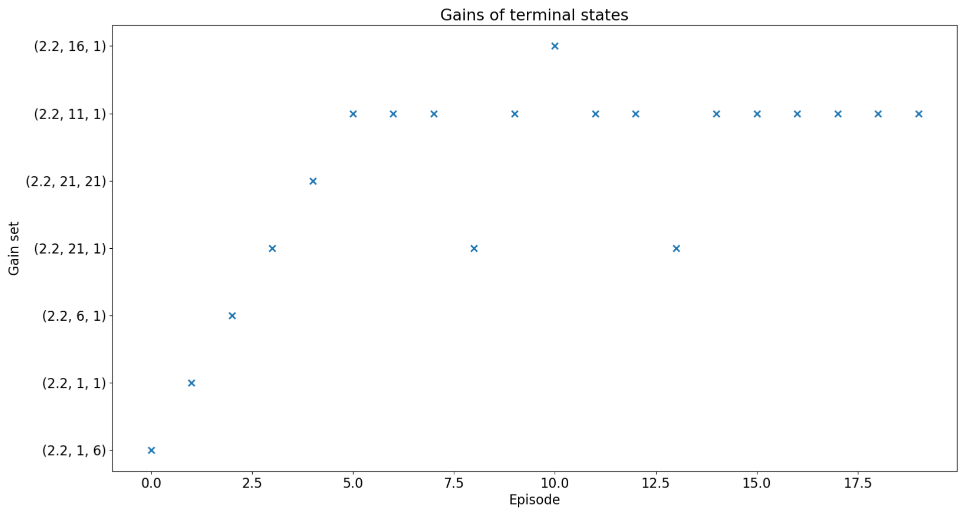
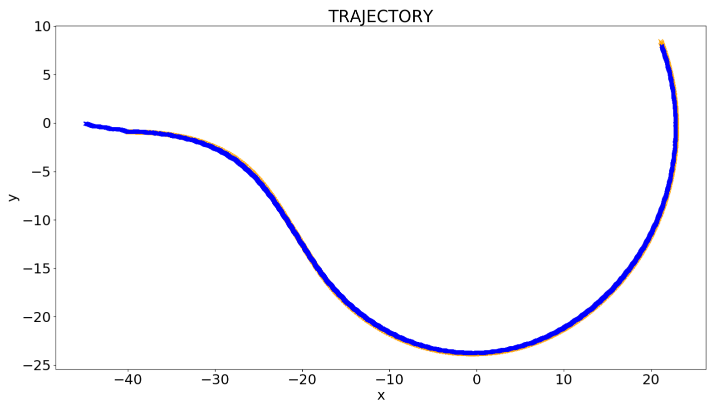


Figure 4.11: Roundabout training: gains of all the terminal states

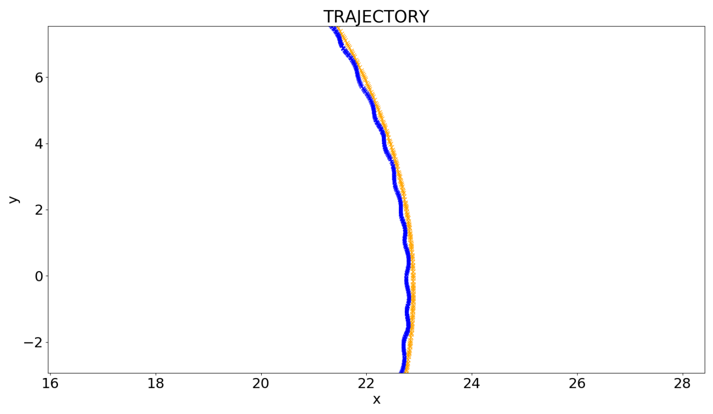
Once again, the agent starts by reaching the terminal state with less efficient sets of gains, improving those gains until converging to $(2.2, 11, 1)$.

To define the set of gains used in the red zone, all terminal gains from episode 10 onward are considered. The resulting center of mass is $\mathbf{(2.2, 12.5, 1.0)}$. The validation process follows the same model as the one before, evaluating the performance of this set of gains while navigating the roundabout and comparing it with the performance of other sets of gains.

The first test consisted of navigating the roundabout while using the chosen set of gains, for 50 simulated seconds. The vehicle starts in position $(-45.0, 0)$, with the correct orientation. Figure 4.12 shows the trajectory of this test:



(a) Trajectory of the roundabout validation test

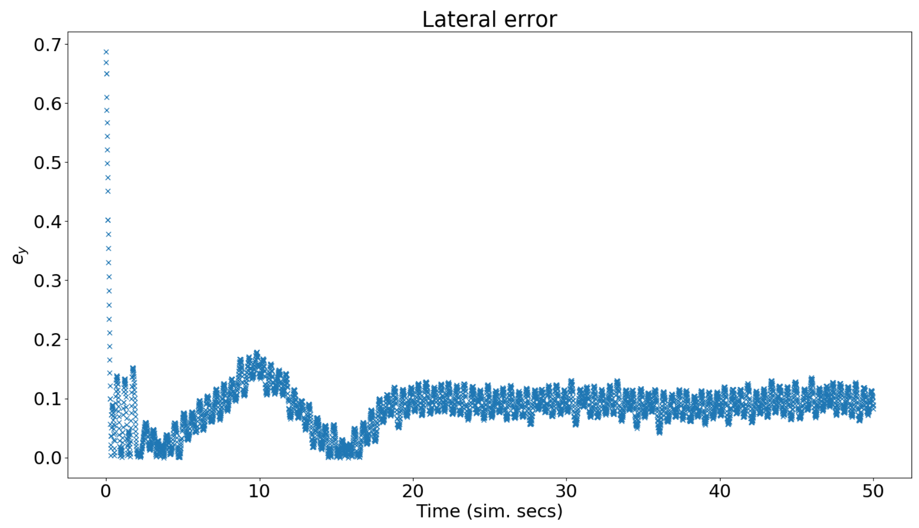


(b) Zoom of the last portion of the trajectory

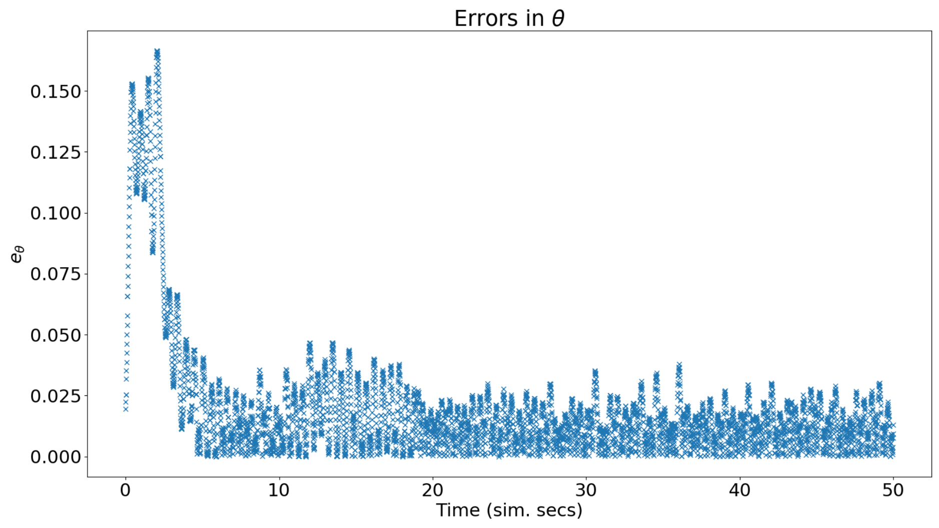
Figure 4.12: Roundabout Validation Test: Trajectory

Figures 4.12 shows the trajectory performed by the vehicle in blue and the reference path in orange. Figure 4.12(a) shows the trajectory and the reference path overlapping, which suggests that the system is able to follow the reference efficiently. Figure 4.12(b) shows the last portion of the roundabout with higher oscillation.

Figure 4.13 shows the absolute values of the lateral and orientation errors registered in this test:



(a) Absolute value of lateral error, e_y

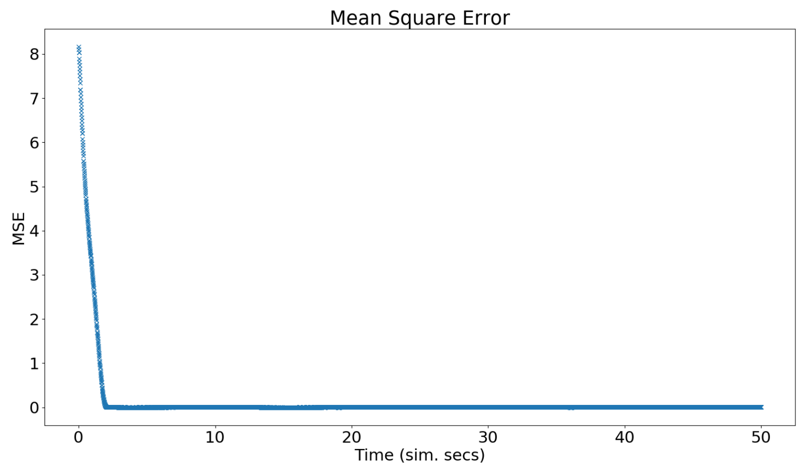


(b) Absolute value of orientation error, e_θ

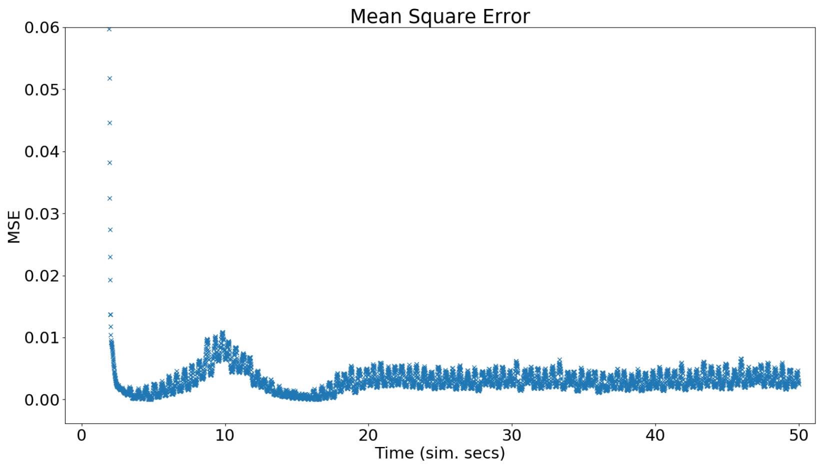
Figure 4.13: Roundabout Validation Test: Lateral and Orientation errors

Figure 4.13(a) shows that the system successfully corrects the 0.9 meter lateral error. After that, it stays always under 0.2 meters of error, oscillating around 0.1 meters of error, except for an initial peek when entering the roundabout. Figure 4.13(b) shows an orientation error spiking at 0.167rad ($\simeq 9.57$ degrees), before entering the roundabout, a slight increase when entering the roundabout (between second 10 and 20),and then stabilizing at around 0.01 rad($\simeq 0.57$ degrees).

Figure 4.14 shows the MSE, calculated as $MSE = \frac{{}^{b}e_x{}^2 + {}^{b}e_y{}^2 + {}^{b}e_\theta{}^2}{3}$, registered during the test. Figure 4.14(a) essentially mirrors the values in figure 4.13(a), with the initial value of 8.2, rapidly dropping and stabilizing in approximately 0.003, as shown in figure 4.14(b).



(a) MSE values



(b) MSE values after the initial error (zoomed)

Figure 4.14: Roundabout Validation Test: Mean Square Error values

Similarly to table 4.2, table 4.3 presents the information gathered about some sets of gains. The values presented were the highest out of 10 samples, and exclude the first 10 seconds of the maneuver. The third column presents the standard deviation, σ, of those 10 samples:

	Average MSE	σ
$(1, 1, 1)$	$8.8\text{e}{-3}$	$3.1\text{e}{-4}$
$(2.2, 1, 1)$	$5.8\text{e}{-3}$	$2.7\text{e}{-4}$
$\mathbf{(2.2, 12.5, 1.0)}$	**$3.0\text{e}{-3}$**	**$1.4\text{e}{-4}$**
$(2.2, 21, 21)$	$6.7\text{e}{-3}$	$2.9\text{e}{-4}$
$(3.4, 1, 1)$	0.0100	$1.9\text{e}{-3}$
$(4.6, 11, 21)$	0.01612	$3.0\text{e}{-3}$
$(5.8, 21, 21)$	0.120	0.0305

Table 4.3: Average MSE of set of gain from the roundabout validation tests

The chosen set of gains presents the lowest MSE value, followed by $(2.2, 1, 1)$ and $(2.2, 21, 21)$. Also, comparing the gains $(1, 1, 1)$ and $(5.8, 21, 21)$, it seems that lower gain values produce lower MSE. Contrary to the previous maneuver, all the sets of gains presented very low standard deviation values. The lowest σ values are again found closer to the chosen gains.

Once again, the chosen gains produce low lateral errors and MSE values. Although sensible to variations in values, table 4.3 suggests that the agent's choice of gains are in the neighborhood of the gains that minimize the error.

4.3 System

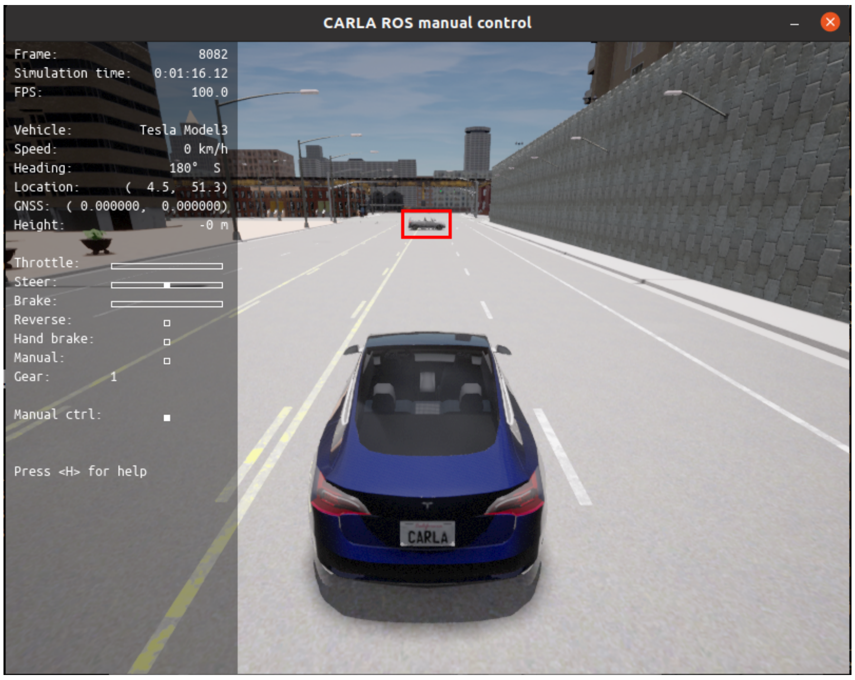
After training the agent and choosing the best gains, the complete system, presented in 3.1, can be tested. The system is expected to have an appropriate behaviour in each of the zones while never exceeding the speed limit imposed: **tuning the gains** in the blue and red zone, **limiting the steering angle range** in the orange zones, and **avoiding a frontal collision** by stopping the vehicle in the green zone.

4.3.1 Collision Avoidance Testing

The collision avoidance method was tested by spawning a second vehicle 40 meters away from the system. The obstacle detector sensor is configured to have the *distance* $d = 20$ (the minimum distance for detection) and the *hit_radius*, $r = 5$ (radius of sphere where it is considered the system hits the vehicle). When the vehicle detects an obstacle, the High Level Controller imposes a new equation for the imposed linear velocity, to substitute equation (3.3), defined as:

$$v = v_i \cdot (d_i/d), \tag{4.6}$$

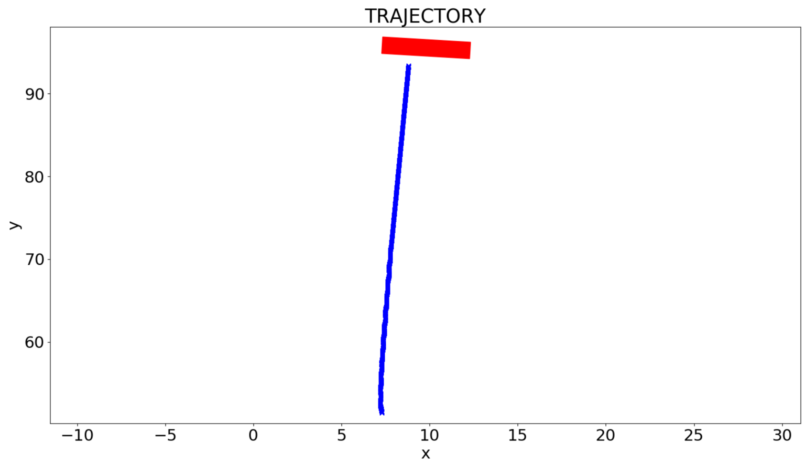
where v_i is the velocity of the vehicle when the obstacle is first detected (20 meters away) and d_i is the distance to the obstacle registered by the sensor at each time-step. Equation (4.6) defines a steady decrease from the initial velocity until it reaches zero, when the vehicle enters the hit area. The High Level Controller also imposes null values for the steering angle, ϕ, and the angular velocity, ω. This sensor is programmed to only detect other vehicles. Figure 4.15 shows the testing environment:



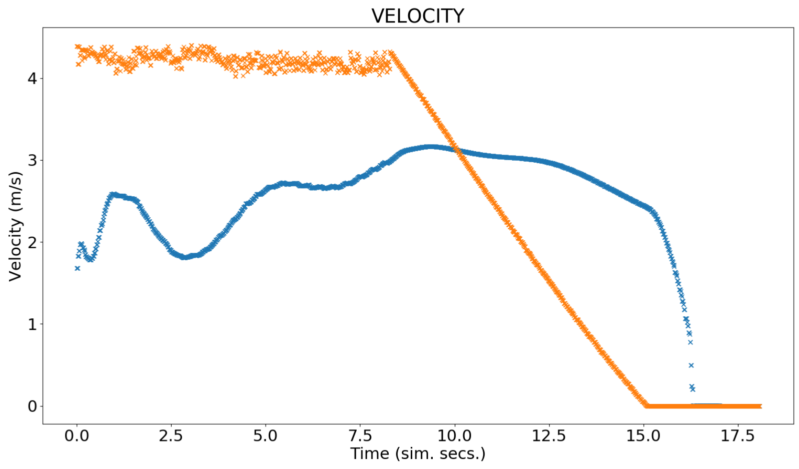
```
Frame:                  8082
Simulation time:  0:01:16.12
FPS:                   100.0

Vehicle:         Tesla Model3
Speed:                0 km/h
Heading:            180°   S
Location:      (  4.5,  51.3)
GNSS:  ( 0.000000,  0.000000)
Height:                 -0 m

Throttle:
Steer:
Brake:
Reverse:                 □
Hand brake:              □
Manual:                  □
Gear:         1

Manual ctrl:             ■


Press <H> for help
```

Figure 4.15: Environment used for testing collision avoidance

Figures 4.16 show the result of the testing. Figure 4.16(a) shows the trajectory of the vehicle, with the other vehicle illustrated as a red rectangle, while figure 4.16(b) shows the velocity imposed in the system by the controller, in orange, and the real velocity of the system, in blue, registered in each second:



(a) Trajectory performed during the test



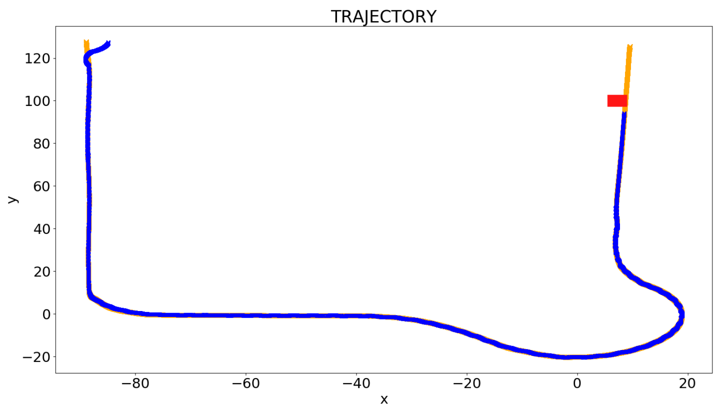
(b) Velocity per simulated sec. during the test

Figure 4.16: Collision avoidance test results

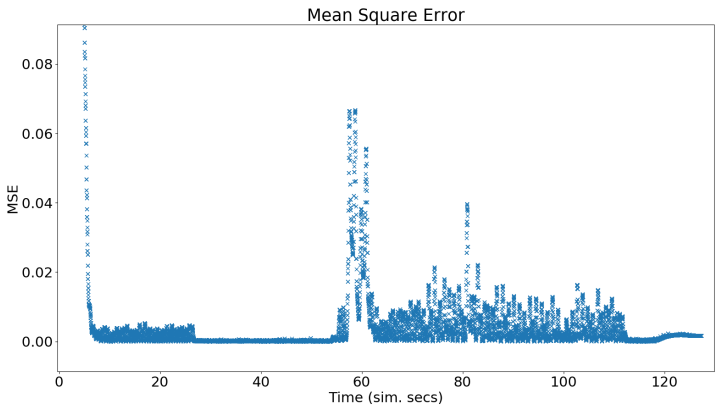
The designed function is working as expected: the vehicle's linear velocity starts decreasing 20 meters before and completely stopping a few meters away from the second vehicle. Figure 4.16(b) shows the decreasing slope of the imposed velocity, calculated by equation (4.6). Starting at around 10 sim. seconds, the system's real velocity (blue) decreases slowly, and then increasingly faster until it peeks to zero at around second 15.5, around the time the imposed velocity (orange) reaches zero, successfully stopping the system 1 sim. second after. Figure 4.16(b) is also a good demonstrator of the simulated vehicle's delay of response to the values imposed by the controller - although the initial imposed velocity is over 4, the vehicle is not able to reproduce this velocity in 10 sim. seconds. Similarly, the vehicle only reaches a full stop approximately 8 sim. secs after the imposed velocity starts to drop.

4.3.2 Full System Testing

The system was tested while navigating in the environment illustrated in figure 3.11, following the reference path defined in red. The chosen gains for the blue and red zones were, respectively, $(\mathbf{2.42, 11.67, 1.33})$ and $(\mathbf{2.2, 12.5, 1.0})$. The system is spawned in position $(-84.9, 127.6056)$ and the second vehicle is spawned in position $(7.6, 100)$. The steering angle range is reduced to $10°$ inside the orange zones. For testing purposes, the speed limit imposed is 4 m/s ($\simeq 14.4$ km/h). The results are presented in the following images:



(a) Trajectory from full system testing



(b) MSE value per time-step during full system test

Figure 4.17: Full system test results

Figure 4.17(a) shows the trajectory performed by the system, in blue, and the reference path, in orange, which are superimposed. The second vehicle is marked by a red rectangle. The figure shows the system successfully follows the reference path, without any major errors or collisions. The first portion of the trajectory shows the lane changing maneuver being executed successfully. The last portion shows the collision avoidance function in action, identifying the second vehicle and coming to a full stop a few meters away from it, despite the reference path. This produces a slight deviation from the reference path, the result of imposing $\omega = 0$ and $\phi = 0$.

Figure 4.17(b) shows the MSE value per time-step graph, zoomed in the portion after the initial error. The test can be divided into portions: the first one, from 0 to 68 sim. secs., includes the blue zone and the left turn that transitions from there to the red zone. From 68 to 119 sim. secs we have the red zone and the transition from there to the green zone. The green zone is the last portion, from 119 onward. During the lane changing maneuver, the figure shows a initial high MSE, followed by a drop of value to below 0.01 (in conformity with the validation test to the lane changing maneuver presented earlier). After entering the first orange zone, it drops even more, below 0.001. After that, the left turn causes these values to peek above 0.06, dropping afterwards. The second portion shows MSE values oscillating around 0.005, similarly to the validation test performed to the roundabout maneuver. The values drop again to less around 0.001 after entering the second orange zone. The peek at the final portion is the result of the slight deviation observed in fig. 4.17(a), caused by the collision avoidance function, stabilizing when the vehicle stops.

Figure 4.18 shows the velocity imposed in the system by the controller, in orange, and the real velocity of the system, in blue, registered in each time-step. The system's delay in reproducing the velocity imposed by the controller is evident once again - looking, for instance, at the first portion, the system's velocity takes about 20 to 30 sim. secs to reach the imposed value. The velocity limit function is working as expected, as it is possible to see in $57 - 85$ sim. secs and again in $102 - 122$ sim. secs., where the imposed velocity stops at 4 m/s. However, the real velocity of the system seems to be able to exceeds this limit, even if ever so slightly, which implies that the imposed limit should be below what is actually desired for the system.

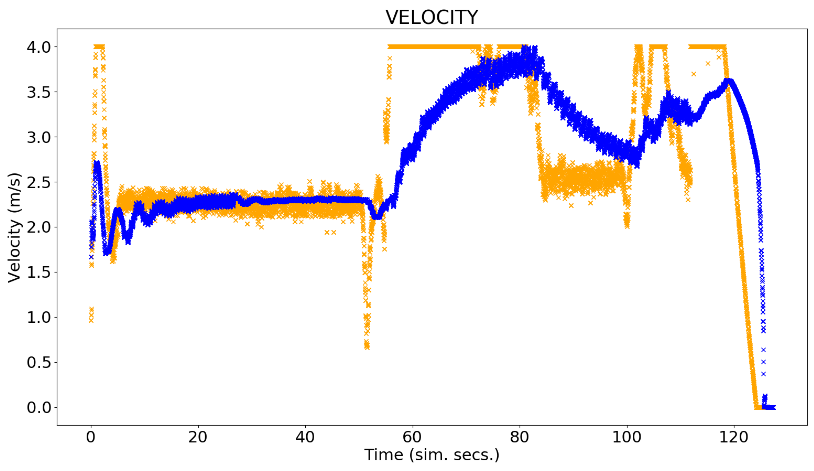


Figure 4.18: Full system test: Imposed velocity (orange) and real velocity of the system (blue) in each time-step

Table 4.4 shows the average MSE of the test performed with the chosen gains and another one performed with gains outside the "preferred cluster". To minimize the value dispersion, the MSE values registered during the final portion (green zone) were ignored:

	Average MSE	σ
$(0.68, 21, 21), (2.2, 21, 21)$	0.0742	$5.5\text{e}{-3}$
$\mathbf{(2.42, 11.67, 1.33), (2.2, 12.5, 1.0)}$	$\mathbf{0.0711}$	$\mathbf{2.0e{-3}}$
$(2.42, 21, 21), (5.8, 21, 21)$	0.164	$1.91\text{e}{-2}$

Table 4.4: Average MSE values of the system test using the chosen sets of gains and sets of gains outside the cluster

The average MSE values produced by the chosen sets of gains are compared with other two sets gains. The third column shows the standard deviation, σ, from the 10 samples registered for each set of gains. The table shows the chosen set of gains produces the lowest average MSE, although with a very little difference in value from the first set of gains. It is worth noting that, comparing to tables 4.2 and 4.3, the differences between average MSE values of each set of gains is very small. This suggests that this type of tuning has more impact when applied locally (in a specific zone or for a maneuver) than generally (throughout the full path).

The results demonstrate all of the proposed functions working correctly. The trajectory in fig. 4.17(a) and the velocity values in fig. 4.18 demonstrates that the system does not engage in unsafe behaviour, like collisions or excessive velocity. Figure 4.17(b) indicates the system is able to consistently follow the reference with very little error. The MSE values in table 4.4 suggest the chosen gains are in the neighbourhood of values that overall minimize the trajectory error.

Table 4.5 presents some reference values taken from the literature for comparison:

Reference values	Values from the Lane Changing maneuver	Values from the Roundabout Navigation maneuver	Values from the complete system test
[41]$RMSE_\theta = 0.042$ [42]$RMSE_\theta = 0.040 - 0.073$ [43]$RMSE_\theta = 0.0709 - 0.2076$	$RMSE_\theta = 0.2543$	$RMSE_\theta = 0.035$	$RMSE_\theta = 0.1211$
[44]$RMSE_{pos.} = 0.9$	$RMSE_{xy} = 1.149$	$RMSE_{xy} = 0.0942$	$RMSE_{xy} = 0.446$
[45]$E_y < 0.25m$ [46]$E_y = 0.075 - 0.15m$	$E_y < 0.47m$	$E_y < 0.18m$	$E_y < 0.45m$

Table 4.5: Reference values taken from the reviewed literature, compared to the values registered in this work

The Root Mean Square Error (RMSE) values shown are calculated as $RMSE_\theta = \sqrt{E_\theta^2}$ and $RMSE_{xy} = \sqrt{\frac{E_y^2+E_x^2}{2}}$, where $E_{x,y,\theta}$ is the average of the error over the duration of the simulation.

According to the table, all of the values from this work are comparable to the reference values. The numbers registered for the lane changing maneuver, on the other hand, are greater than the reference. This is due to the conditions under which the error values are registered for that maneuver: the reference path for this scenario (fig.3.7) does not include the lane changing movement, and instead only includes a path following a straight road. For that matter, the system registers high values of error until the maneuver is complete, resulting in greater average RMSE and MSE values.

It is also worth noting that the values for the roundabout navigation are better than the values registered during the test to the complete system, highlighting the idea that this type of tuning is better suited to the maneuvers, as intended.

Chapter 5

Conclusion

This work proposes a different approach for using a RL algorithm in an autonomous driving system, comparable to how humans learn to drive: instead of the common fast learning with online tuning, the algorithm requires an entire maneuver to be executed to gather information and learn from it, much like a person driving in a new environment would. The drawback of using this offline tuning is the training times. A maneuver can take a few seconds to complete, which, in a simulated environment, can translate to a few hours per episode. Given that, the proposed system implements an offline ”long-term” gain tuning, that would be done throughout several weeks, months or years. Every time the system performs a maneuver, in a given environment, it registers data and trains the algorithm. The tuned gains are later updated and used the next time. This approach not only avoids dangerous online fluctuations of the gains but also introduces a way to adjust the gains to specific maneuvers or zones inside a city.

Meeting the Q-learning convergence conditions, combined with the learning curves in fig. 4.5 and fig. 4.10, suggest a proper implementation of the algorithm. The validation tests performed to the chosen gains suggest that the algorithm can tune the gains to values that minimize the trajectory error. The results in section 4.3 show that all the modules of the system are working as intended. It is worth noting that, after the training, the sets of gains selected to perform each maneuver presented values very close to each other. This poses the idea that, within a certain grain of tuning, a single set of gains is adequate to any maneuver.

During this project, RL has shown to be a good approach when working with the CARLA simulator and limited computational resources. By not requiring labeled datasets or image processing, using RL significantly reduced the computational requirements, while still providing adaptability and robustness to the system. Using a simplified version of the Q-learning algorithm also helped surpass the shortage of computational resources, while seemingly not jeopardizing the viability of the agent. Although finding the best reward function was a long trial-and-error process, experiments made throughout this work suggested that multiple functions could turn out similarly good results.

CARLA offers a vast range of tools to develop self-driving autonomous vehicles. Being an open-source project, it also provides a good support network for problem-solving and software improvement. Working with the CARLA simulator, however, proved to be one of the biggest challenges of this thesis. Configuring it to perform the necessary tasks was a slow and difficult process, but the biggest setback was the dispersion observed in the data acquired. This compromised the performance of the agent. Solutions to this problem are proposed in the next section.

5.1 Future Work

These are the proposed future works:

- The reviewed literature leads to believe that the system would benefit from having a neural network - as explained in 2, hybrid learning methods improve learning speeds and provide better adaptation to new environments. The network can be used as a function approximator of the Q-learning algorithm, accelerating the Q-learning training process, but also adding an extra layer of complexity to the algorithm, as well as requiring time to train [35]. Despite that, performance may increase after training.

  A more interesting usage of a Neural Network in this system, however, would be to identify the map zone the vehicle is currently in. Instead of relying on the vehicle location, the proposed improvement would have a neural network process RBG images of the vehicle's surroundings and identify traits such as roundabouts or straight roads with no obstacles, which would trigger a change in the controller gains calculated by the RL agent;

- One of the biggest obstacles faced was the dispersion observed in the data acquired from the simulator. Training and testing a Q-learning agent in such conditions may compromise the performance of the agent. Exploring longer in a vaster action space may help solve this issue. This is achieved by lowering the learning rate and increasing the grain of gain tuning;

- As mentioned in section 2.4, an interesting upgrade to the Q-learning algorithm would be the Double Q-learning algorithm - as suggested in [47], this algorithm might improve the performance of the agent in a stochastic environment such as the CARLA simulator, with the only downside of doubling the memory requirements;

- Real-world application is the most important goal of this work. The vehicle performed the maneuvers and followed the reference path with relatively little error, as shown in figures 4.7 and 4.12, while also maintaining safe velocity values (under 15 km/h), despite the limitations imposed by the simulator. Some of these limitations, like client-server latency, do not exist in real-world applications. For these reasons, a successful and safe transition to a physical vehicle is expected. However, some adaptations are required for this transition, namely changing the gain and state limits, the steering angle range and adapting the system to work in a new physical map;
- During testing, the system presented average velocities of under 25 km/h. Althought this is preferred for early testing in the real world, more advanced real-world implementations might require higher average velocities. Future work would be to add a velocity component to the Reward function of the Q-Learning algorithm, so as to have control on the average velocity of the vehicle.

Bibliography

[1] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “Carla: An open urban driving simulator,” in *Conference on robot learning*. PMLR, 2017, pp. 1–16.

[2] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, “Ros: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[3] K. Bimbraw, “Autonomous Cars: Past, Present and Future - A Review of the Developments in the Last Century, the Present Scenario and the Expected Future of Autonomous Vehicle Technology,” in *Proceedings of the 12th International Conference on Informatics in Control, Automation and Robotics*. SCITEPRESS - Science and and Technology Publications, 2015.

[4] J. Delcker, “The man who invented the self-driving car (in 1986),” 7 2018. [Online]. Available: https://www.politico.eu/article/delf-driving-car-born-1986-ernst-dickmanns-mercedes/

[5] Y. Lecun, E. Cosatto, J. Ben, U. Muller, and B. Flepp, “Dave: Autonomous off-road vehicle control using end-to-end learning,” *DARPA-IPTO Final Report*, 2004.

[6] A. Forrest and M. Konca, “Autonomous cars and society,” *Worcester Polytechnic Institute*, 2007.

[7] S. S. Shadrin and A. A. Ivanova, “Analytical review of standard sae j3016 ≪taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles≫ with latest updates,” *Avtomobil’. Doroga. Infrastruktura.*, no. 3 (21), p. 10, 2019.

[8] Luigi Glielmo, “Vehicle-to-Infrastructure Control: Grand Challenges for Control,” IEEE Control Systems Society, Tech. Rep., 2011. [Online]. Available: www.ieeecss.org.

[9] W. Pitts and W. S. McCulloch, “How we know universals the perception of auditory and visual forms,” *The Bulletin of Mathematical Biophysics*, vol. 9, no. 3, 9 1947.

[10] D. Hebb, *The Organization of Behavior*. Psychology Press, 4 2005.

[11] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3-4, 5 1992.

[12] C. F. Higham and D. J. Higham, “Deep learning: An introduction for applied mathematicians,” 2018.

[13] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018.

[14] C. Thorpe, M. Herbert, T. Kanade, and S. Shafer, “Toward autonomous driving: the CMU Navlab. I. Perception,” *IEEE Expert*, vol. 6, no. 4, 8 1991.

[15] D. A. Pomerleau, “ALVINN: AN AUTONOMOUS LAND VEHICLE IN A NEURAL NETWORK,” CARNEGIE-MELLON UNIV PITTSBURGH PA ARTIFICIAL INTELLIGENCE AND PSYCHOLOGY, Tech. Rep., 1989.

[16] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda, “A Survey of Autonomous Driving: Common Practices and Emerging Technologies,” *IEEE Access*, vol. 8, pp. 58 443–58 469, 2020.

[17] V. Rausch, A. Hansen, E. Solowjow, C. Liu, E. Kreuzer, and J. K. Hedrick, “Learning a deep neural net policy for end-to-end control of autonomous vehicles,” in *2017 American Control Conference (ACC)*. IEEE, 5 2017.

[18] F. Rosique, P. J. Navarro, C. Fernández, and A. Padilla, “A Systematic Review of Perception System and Simulators for Autonomous Vehicles Research,” *Sensors*, vol. 19, no. 3, 2 2019.

[19] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, “A survey of deep learning techniques for autonomous driving,” *Journal of Field Robotics*, vol. 37, no. 3, 4 2020.

[20] S. Kuutti, R. Bowden, Y. Jin, P. Barber, and S. Fallah, “A Survey of Deep Learning Applications to Autonomous Vehicle Control,” *CoRR*, 12 2019. [Online]. Available: http://arxiv.org/abs/1912.10773

[21] H. Xu, Y. Gao, F. Yu, and T. Darrell, “End-to-end learning of driving models from large-scale video datasets,” in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-January. Institute of Electrical and Electronics Engineers Inc., 11 2017, pp. 3530–3538.

[22] H. M. Eraqi, M. N. Moustafa, and J. Honer, “End-to-End Deep Learning for Steering Autonomous Vehicles Considering Temporal Dependencies,” *arXiv preprint arXiv:1710.03804*, 2017.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[24] A. Sallab, M. Abdou, E. Perot, and S. Yogamani, “Deep Reinforcement Learning framework for Autonomous Driving,” *Electronic Imaging*, vol. 2017, no. 19, 1 2017.

[25] A. G. Barto and S. Mahadevan, “Recent Advances in Hierarchical Reinforcement Learning,” *Discrete Event Dynamic Systems*, vol. 13, no. 4, 2003.

[26] H. Hasselt, “Double Q-learning,” in *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., vol. 23. Curran Associates, Inc., 2010. [Online]. Available: https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf

[27] D. Zhao, B. Wang, and D. Liu, “A supervised Actor–Critic approach for adaptive cruise control,” *Soft Computing*, vol. 17, no. 11, 11 2013.

[28] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, “Imitation Learning,” *ACM Computing Surveys*, vol. 50, no. 2, 6 2017.

[29] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “Ssd: Single shot multibox detector,” in *European conference on computer vision*. Springer, 2016, pp. 21–37.

[30] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[31] S. Wang, D. Jia, and X. Weng, “Deep Reinforcement Learning for Autonomous Driving,” *arXiv preprint arXiv:1811.11329*, 11 2018.

[32] X. Xiong, J. Wang, F. Zhang, and K. Li, “Combining deep reinforcement learning and safety based control for autonomous driving,” *arXiv preprint arXiv:1612.00147*, 2016.

[33] M. Bojarski, D. Del 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,” *arXiv preprint arXiv:1604.07316*, 2016.

[34] S. Bari, S. S. Zehra Hamdani, H. U. Khan, M. u. Rehman, and H. Khan, “Artificial neural network based self-tuned pid controller for flight control of quadcopter,” in *2019 International Conference on Engineering and Emerging Technologies (ICEET)*, 2019, pp. 1–5.

[35] P. Kofinas and A. I. Dounis, “Fuzzy Q-learning agent for online tuning of PID controller for DC motor speed control,” *Algorithms*, vol. 11, no. 10, 10 2018.

[36] Q. Shi, H. K. Lam, B. Xiao, and S. H. Tsai, “Adaptive PID controller based on Q-learning algorithm,” *CAAI Transactions on Intelligence Technology*, vol. 3, no. 4, 2018.

[37] Epic Games, “Unreal engine.” [Online]. Available: https://www.unrealengine.com

[38] C. Clapham and J. Nicholson, *The Concise Oxford Dictionary of Mathematics*. Oxford University Press, 1 2009.

[39] Epic Games, "SphereTraceByChannel." [Online]. Available: https://docs.unrealengine.com/4.26/en-US/BlueprintAPI/Collision/SphereTraceByChannel/

[40] E. Even-Dar and Y. Mansour, "Learning rates for Q-learning," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2111, 2001.

[41] R. Valiente, M. Zaman, S. Ozer, and Y. P. Fallah, "Controlling steering angle for cooperative self-driving vehicles utilizing cnn and lstm-based deep networks," in *2019 IEEE intelligent vehicles symposium (IV)*. IEEE, 2019, pp. 2423–2428.

[42] G. Katz, A. Roushan, and A. Shenoi, "Supervised learning for autonomous driving," 2017.

[43] S. Du, H. Guo, and A. Simpson, "Self-driving car steering angle prediction based on image recognition," *arXiv preprint arXiv:1912.05440*, 2019.

[44] S. M. Grigorescu, B. Trasnea, L. Marina, A. Vasilcoi, and T. Cocias, "Neurotrajectory: A neuroevolutionary approach to local state trajectory learning for autonomous vehicles," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3441–3448, 2019.

[45] L. Yu, X. Shao, Y. Wei, and K. Zhou, "Intelligent land-vehicle model transfer trajectory planning method based on deep reinforcement learning," *Sensors*, vol. 18, no. 9, p. 2905, 2018.

[46] C. J. Ostafew, A. P. Schoellig, and T. D. Barfoot, "Robust constrained learning-based nmpc enabling reliable mobile robot path tracking," *The International Journal of Robotics Research*, vol. 35, no. 13, pp. 1547–1563, 2016.

[47] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, "Q-Learning Algorithms: A Comprehensive Classification and Applications," *IEEE Access*, vol. 7, pp. 133 653–133 667, 2019.

[48] S. J. S. J. Russell and P. Norvig, *Artificial intelligence : a modern approach*. Prentice Hall, 1995.

[49] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[50] Z. Ghahramani, "Unsupervised learning," 2004.

[51] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. Yogamani, and P. Perez, "Deep Reinforcement Learning for Autonomous Driving: A Survey," *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[52] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.

[53] CARLA Team, "CARLA Documentation," 2021. [Online]. Available: https://carla.readthedocs.io/en/0.9.11/

[54] ——, "ROS Bridge Documentation," 2021. [Online]. Available: https://carla.readthedocs.io/projects/ros-bridge/en/latest/

Appendix A

Table of Concepts

The following table concatenates the most relevant concepts in the areas of Autonomous Driving systems and Machine Learning, with a succinct definition for each of them. This table has the purpose of giving the necessary context to the concepts used throughout the work.

Table A.1: Most relevant concepts on Autonomous Driving Systems and Deep Learning

Concept	Description
Agent	An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors [48]
Neural Networks	Computing system used in machine learning. Their architecture is based on interconnected nodes. A neural network architecture and functioning is inspired by the biological neural networks present on animal's brains
Deep Learning	Machine Learning area concerning algorithms inspired by or using artificial neural networks
Supervised Learning	Machine learning method based on labeled datasets. The agent learns by observing several data examples and associated labels and then tries to predict those labels for new examples [49]. In simple terms, the process goes as follows: the network predicts a label for an example. Then, it compares its prediction with the predefined label and, by taking into account the error between the two, it adjusts its parameters to make a better prediction. Eventually, it is able to correctly predict labels even from unlabeled examples
Unsupervised Learning	Machine learning method that receives input data but, instead of obtaining target outputs or rewards from the environment, it finds patterns in that input data. Examples of Unsupervised Learning (UL) include clustering and dimensionality reduction [50]

Convolutional Neural Networks	A class of feed forward neural network is mainly used for processing spatial information, such as images, and can be viewed as an image feature extractor [19]. It has specific characteristics that differ it from the other classes of neural networks, such as its shared-weight architecture and translation invariance
Recurrent Neural Networks	A class of neural networks that is especially good in processing temporal sequence data, such as text or video streams [19]. They differ from other neural networks in the sense that they contain a time-dependent feedback loop in their memory cell. This allows them to exhibit temporal dynamic behavior, which makes them a good fit for the mentioned applications
Long Short-Term Memory	A specific Recurrent Neural Network architecture that, unlike traditional RNN, has feedback connections. LSTM are non-linear function approximators for estimating temporal dependencies in sequence data [19]
Inverse Reinforcement Learning	It is a subset of Reinforcement Learning [51] based on Imitation Learning, in which the reward function is not specified, but the agent attempts to learn it from experts' demonstrations.This is an approach for solving the problem of extracting a reward function given observed optimal behavior
Value-based algorithm	It is a type of algorithm used in the context of reinforcement learning to find the optimal policy for an agent. It estimates the value function, which represents the value (reward) of being in a state [20]
Policy gradient algorithm	It is a type of Reinforcement Learning algorithm used to find the optimal policy for an agent. The algorithm is based on parameterizing the policy function and then updating those parameters to maximize the reward [20]
Actor-Critic algorithm	It is a type of Reinforcement Learning algorithm used to find the optimal policy for an agent. It is a hybrid method that combines the benefits of both value-based and policy gradient algorithms. Basically, it divides the model in two:one part chooses which action the agent should take (actor) and another part evaluates the actions taken and communicates how they should adjust (critic). Both parts are training simultaneously
Advantage Actor Critic	It is an actor-critic algorithm where the Critic part learns the Advantage values instead of the Q values. Advantage functions are a part of the Q function that represent how better an action is compared to others
Asynchronous Advantage Actor Critic	It is an actor-critic algorithm similar to Advantage Actor Critic but it consists of multiple independent agents who interact with a different copy of the environment in parallel, exploring much more of the environment in a shorter time [52]. The agents update a global network periodically and asynchronously, and after updating they all reset their parameters to the ones in the global network

Deep Q-Network	It is a network that incorporates a variant of the Q-learning algorithm, by using a deep neural network as a non-linear Q-function approximator over high-dimensional state spaces [51]. A common implementation is to initialize a table of possible state-action combinations (q-table) and then use a neural network to predict based on the q-table,while simultaneously updating its values based on those predictions

Appendix B

Installing and configuring the CARLA simulator

The CARLA documentation website [53] offers detailed information about the simulator. It also offers a step-by-step on how to install it and the system requirements. The ROS bridge was installed subsequently by following the installation instructions on the ROS Bridge Documentation website [54]. This work uses the CARLA version ”0.9.9-4” and the ROS bridge version ”0.9.10.1”. It is, however, recommended to install a ROS bridge version that is compatible with the CARLA version, which is not the case for the two versions used. The simulation is run on a Ubuntu 20.04 (Focal) operating system (OS) and ROS Noetic, version 1.15.11. Both the CARLA and the ROS bridge were installed from the source Repository. The simulator's software is constantly being corrected and improved, so it is highly recommended to run the simulator in the latest versions and the most recent OS and ROS versions.

B.1 Configuring the CARLA simulator

Table B.1 shows a list of files that can be used to configure the simulator parameters:

File path	Parameters
[HOME]/carla-ros-bridge/catkin_ws/src/ros-bridge/ carla_ros_bridge/test/settings.yaml	*fixed_delta_seconds, synchronous_mode_wait_for_vehicle_control_command, synchronous_mode, vehicle_filter*
[HOME]/carla-ros-bridge/catkin_ws/src/ros-bridge/ carla_ackermann_control/config/settings.yaml	*speed_Kp, speed_Ki, speed_Kd, accel_Kp, accel_Ki, accel_Kd, min_accel*
[HOME]/carla-ros-bridge/catkin_ws/src/ros-bridge/carla_ros_bridge/ launch/carla_ros_bridge_with_example_ego_vehicle.launch	*vehicle_filter, spawn_point, fixed_delta_seconds, synchronous_mode_wait_for_vehicle_control_command*
[HOME]/carla-ros-bridge/catkin_ws/src/ros-bridge/ carla_spawn_objects/config/objects.json	Vehicle sensor setup (json format)

Table B.1: Table of file paths and parameters configured in each one

The parameters values are set as follows:

1. *synchronous_mode_wait_for_vehicle_control_command* and *synchronous_mode* are set to ”True”;
2. *fixed_delta_seconds* is set to 0.01;
3. Ackermann controller gains are set to default values;
4. *vehicle_filter* defines the model of the vehicle spawned, and is defined as ”tesla.model3”;
5. *spawn_point* defines the initial pose of the vehicle (see Chapter 3 for specific values);
6. The vehicle sensor setup is defined as explained in chapter 3.1, following the *.json* file format.