Heriot-Watt University

Masters Thesis

Evaluating the Performance of Different Reinforcement Learning Methods for Autonomous Racing

Author:
Henri-Louis Boisvert

Supervisor:
Dr. Ekaterina Komendantskaya

A thesis submitted in fulfilment of the requirements for the degree of MSc in Artificial Intelligence

in the

School of Mathematical and Computer Sciences

11/09/2022

Declaration of Authorship

I, Henri-Louis Boisvert, declare that this thesis titled, 'Evaluating the Performance of Different Reinforcement Learning Methods for Autonomous Racing' and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: *Henri-Louis Boisvert*

Date: *11/09/2022*

Abstract

The increasing popularity of Reinforcement Learning over the last decade has transformed the AI field. One area that has benefited from this so-called deep learning revolution is autonomous driving and more specifically autonomous racing. Many Reinforcement Learning based controllers have been implemented with varying levels of success, to such an extent that it has become a challenge to find the method with the right compromise between performance and complexity of implementation. Furthermore, most implementations of Reinforcement Learning controllers for autonomous racing are trained in a simulated environment and assume that the Sim2Real gap won't have too much of an impact on performance: this is something that can have a considerable impact on the choice of a specific method for real-life applications.

This project will have a double aim. Firstly, it will be to compare the performance of different autonomous racing controllers and their ability to generalise to new data to help users choose one that corresponds to their needs. Secondly, it will be to compare the Sim2Real gaps to give users an idea of performance in real-life situations; this will be achieved by using the F1Tenth platform. This research report details my review of the relevant literature and establishes research questions, hypotheses, objectives, and a well-defined experimental protocol.

Acknowledgements

I would like to thank my project supervisor Dr Ekaterina Komendantskaya for the time she spent providing me with guidance and advice. I would also like to thank the other members of the Lab for Artificial Intelligence Verification (LAIV), especially Luca Arnaboldi and Marco Casadio for sharing their technical knowledge on Reinforcement Learning and the F1Tenth system.

Contents

List of Figures

List of Tables

Abbreviations

RL	**R**einforcement **L**earning
DRL	**D**eep **R**einforcement **L**earning
NN	**N**eural **N**etworks
CNN	**C**onvolutional **N**eural **N**etwork
ROS	**R**obot **O**perating **S**ystem
DQN	**D**eep **Q**-**N**etwork
DDPG	**D**eep **D**eterministic **P**olicy **G**radient
FTG	**F**ollow **T**he **G**ap
PID	**P**roportional-**I**ntegral-**D**erivative
VM	**V**irtual **M**achine
LiDAR	**Li**ght **D**etection **A**nd **R**anging
SVM	**S**upport **V**ector **M**achine

Chapter 1

Introduction

1.1 Motivation and research questions

For the last decade, the driving industry has seen steady progress in implementing autonomous-vehicle technology, with companies such as Tesla, Waymo, Ford, Baidu and Apple aiming to develop self-driving vehicles for consumers. With that would come significant benefits: improved personal safety, reduced environmental impact, decreased transportation costs, and time-saving (Barabás et al. [2017]). It is estimated that by 2030 a considerable number of driverless vehicles will be travelling on the roads, even though the industry faces significant technological, legal and ethical challenges (Barabás et al. [2017]). Autonomous driving systems rely on sensors to acquire data about their environment, for example, the position of other vehicles or pedestrians. Some of the companies working on self-driving technology have been relying on Light Detection and Ranging (LiDAR) sensors, which use laser beams to map the 3D environment of the vehicle by measuring the difference between the generated and reflected laser beam; the LiDAR then outputs a 3D point cloud corresponding to the scanned environment (Li and Ibanez-Guzman [2020]).

The Machine Learning (ML) field has recently gained a lot of interest for several reasons, notably theoretical developments such as Deep Neural Networks (NNs) and Reinforcement Learning (RL), as well as the increasing availability of large datasets and parallel computing hardware (Gerrish [2018]). RL is an area of machine learning concerned with problems in which an intelligent agent is evolving in its environment while analysing the consequences of its actions thanks to the reward given by the environment. RL

problems can be formulated as Markov Decision Processes (MDPs) if the environment is fully observable or as partially observable MDPs if the agent only has access to a subset of states, as it would be for an autonomous vehicle. The agent's goal is then to find the optimal policy of the MDP, which maximises the expected cumulative reward. Despite its perceived usefulness and growing popularity, Reinforcement Learning has not been successfully applied to commercial autonomous driving, even though several approaches have been developed like in Navarro et al. [2020] and Ahmad El Sallab [2017]. The field of RL has dramatically evolved since Watkins [1989], with the new approach of deep reinforcement learning combining classic reinforcement algorithms with convolutional neural networks, which looks very promising for autonomous driving applications (Ivanov et al. [2020]).

The F1Tenth Autonomous Vehicle System is a modular open-source platform designed by an international community of researchers in 2016 at the University of Pennsylvania for autonomous systems research and education (O'Kelly et al. [2019]). This standardised 1/10 scale autonomous racing car can be equipped with different sensors and performs intensive computations online using a GPGPU Computer platform. It features realistic dynamics and hardware/software capabilities similar to a full-scale platform. The software architecture is based on the Robot Operating System (ROS), a set of software libraries used to build robot applications. The architecture is divided into the perception, planning and control modules, making the F1Tenth platform modular enough to be used for research in fields such as communication systems, robotics, autonomous driving and reinforcement learning. Furthermore, the platform provides a simulator able to run the same code as the car itself, which enables the user to develop a controller in a safe and controlled environment to avoid damaging the car. However, this could be problematic if the controller is incapable of accounting for the sensor noise and the actuators' lack of precision which is unavoidable in the physical setup.

Over the past decades, many RL controllers have been successfully introduced in the autonomous racing field, for example, by Balaji et al. [2020] and Fuchs et al. [2021], so much that, in our opinion, it has become complicated to choose which method to use, with a trade-off between performance and complexity. In our opinion, comparing the efficiency of some of the most popular autonomous racing controllers would be interesting. In this dissertation, we will thus evaluate the efficiency of Deep RL methods over RL controllers, the Wall Following method, and human control of an F1Tenth platform using data from the onboard LiDAR. We will first implement basic RL algorithms like

Q-Learning, and then move on to more advanced controllers. We thus propose the two following research questions:

Research question 1: Does Deep Reinforcement Learning provide a real advantage for controlling an autonomous racing car over less sophisticated methods like Wall Following, according to the selected metrics?

Research question 2: Are Reinforcement Learning controllers robust enough to overcome the simulation to real-life gap? We define robustness as the ability to perform well (according to our metrics) even though the simulated environment may differ from real life.

1.2 Aims and Objectives

This dissertation will compare the viability of different reinforcement learning approaches to develop an autonomous racing controller for an F1Tenth car. We will determine if DRL shows a real improvement over simpler algorithms, like the Wall Following method, which uses LiDAR data and a PID controller to steer the car. In the literature, DRL controllers have been shown to achieve super-human capabilities in the racing game Gran Turismo (Fuchs et al. [2021]), and controllers based on the Deep Deterministic Policy Gradient (DDPG) and Twin Delayed DDPG (TD3) algorithms were successfully implemented on the F1Tenth platform (Ivanov et al. [2020]). The controller applies a control action during training and looks at the reward received. As the training continues, the algorithm would try to maximise the reward by looking at the state space and experimenting with different control outputs. The Neural Network would take the raw LiDAR measurements as input and output the steering for the F1Tenth car. This general goal can be subdivided into smaller objectives:

- **Objective 1:** Define an appropriate reward function which incentivises safe, smooth and fast control of the F1Tenth car for Reinforcement Learning algorithms.
- **Objective 2:** Implement several controllers using different algorithms and Neural Networks (NNs) architectures. In the case of Deep Reinforcement Learning algorithms, the NNs are used to approximate the optimal policy of the racing car, as will be explained in Chapter 2.

- **Objective 3:** Train the controllers in the simulation using different methods, starting with Deep Q-Learning and then moving to a Deep Deterministic Policy Gradient (DDPG) to approximate the Q function using a Bellman equation; test the controllers on different race tracks.
- **Objective 4:** Use the NN controllers trained in the simulator on the physical car and observe the gap between the simulation and the real world; control the car manually with the joystick.
- **Objective 5:** Conclude on the viability of the different algorithms and the superiority (or not) of DRL methods.

The metrics used for evaluating the performance of each controller on several racing tracks will be introduced in Section 3.6.

The different racing tracks will be chosen to evaluate the performance of the controller to generalise to other situations; they consist of the RedBull Racing Track, the Circuit de Monaco, the Silverstone track and a custom-designed oval-shaped track; they can be found in Appendix A. The first three tracks were scaled down to one-tenth of their real size to fit with the F1Tenth system.

Results and Project Scope

This project's main contribution is to offer a better understanding of the capabilities and limitations of different RL algorithms for autonomous racing, especially regarding the gap between the simulation and the real world. Before this project, many RL methods have already been used for autonomous racing; however, the Sim-to-Real gap is often not quantified in detail. We will limit ourselves to comparing only a few algorithms, and available computational capabilities will limit training time. Furthermore, we will restrict our data acquisition to LiDAR, won't use any sensor fusion and will assume the LiDAR data is perfectly accurate.
Extrapolation of our results to more state-of-the-art racing platforms and algorithms will be left for future work.

Chapter 2

Background and Literature Review

This chapter starts by introducing the essential technical and conceptual aspects of RL in Section 2.1. Those basic concepts will then be used to build our controller implementation in the following chapters of this dissertation. In Section 2.2, we will then look at the literature regarding model-free RL algorithms in both discrete and continuous action space. Next, in Section 2.3, we introduce the main concepts of convolutional neural networks, which are one of the main components of our Deep RL controllers. Finally, in Section 2.4 we will look at how to apply those algorithms to autonomous racing using ROS and the F1Tenth system and simulator.
Our goal in this chapter is to start by explaining the basic ideas of RL and NNs to be then able to explain the more sophisticated algorithms that will be implemented in ROS on the F1Tenth platform.

2.1 Reinforcement Learning essentials

2.1.1 What is Reinforcement Learning

Reinforcement Learning is a subfield of Machine Learning concerned with the problem of an agent that must learn behaviour through trial and error interactions with a changing

environment. RL is one of four main ML paradigms (Alloghani et al. [2020]):

- **Unsupervised learning** is a machine learning paradigm concerned with grouping data and finding patterns without any label provided. It is mainly used for solving problems related to clustering, compressing and filtering data and estimating statistical distributions.
- Different from unsupervised learning, **supervised learning** relies on labelled data to perform classification and prediction using various algorithms such as regressions, Bayesian networks and decision trees. The inconvenience with supervised learning is that the training dataset needs to be labelled, which is a very time-consuming task.
- **Semi-supervised learning** is a compromise between supervised and unsupervised learning: the training dataset comprises a majority of unlabelled data samples with a small number of labelled data samples. It is used when the cost of labelling the whole dataset for supervised learning is too high, but having some labelled data offers a significantly improved learning accuracy over unsupervised learning.
- Finally, **reinforcement learning** is the paradigm we will look at more in detail in the following paragraphs.

The increasing popularity of RL has been attributed mainly to the so-called deep learning revolution that started in 2012 and has since transformed the AI field (Sejnowski [2018]). The RL paradigm is presented in Figure 2.1. At each time step t, an agent is changing the state of its environment through an action a_t, and thus finds itself in a new state s_{t+1} after receiving the reward r_{t+1}. The agent has its own representation of the environment, which may be different from the actual state of the environment and follows an action selection policy.

In many ways, reinforcement learning differs from the other main ML field of supervised learning. The main difference is that the agent does not need to be told what action would have been optimal after making an action. This is why RL can be used in several real-world situations when the optimal actions are unknown. Furthermore, unlike supervised learning, the RL agent can face a dynamic environment.

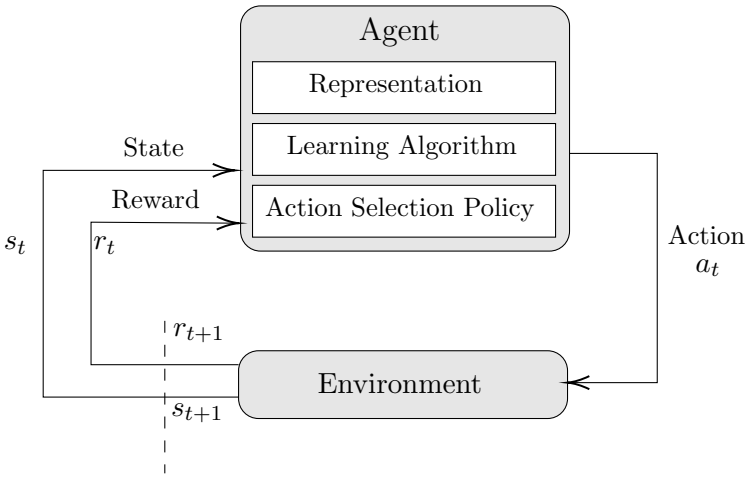

FIGURE 2.1: Reinforcement Learning Paradigm[1]

[1] Unless stated otherwise, figures are my own work

The main categories of RL agents are presented in Figure 2.2. The fundamental distinction is whether or not the agent is given (or learns) a model of the environment, meaning a function predicting rewards and state transitions. A famous example of a model given RL algorithm is the AlphaGo/AlphaZero program, based on the Monte Carlo Tree Search method (MCTS), published in Silver et al. [2016]. However, apart when the agent plays a board game with simple rules, there is no environment model available for the agent, which means that the model has to be learned from experience: those algorithms are model learned. One of the issues with that method is that the agent could behave very well in his learning environment but terribly in the real one because of biases appearing during learning. An example of a model learned method is the Imagination-Augmented Agent (I2A), introduced in Weber et al. [2017].

The alternative to model-based is model-free: those methods tend to be more popular than model-based ones because they are often more easy to tune and implement.

It is important to note that model-based and model-free only refer to whether or not the agent uses predictions of the environment response (next reward and state) for the training. Those predictions could be either computed outside of the training algorithm, for example, by a program that understands the rules of a board game (model given) or learned by the agent in an approximate way (model learned). In the case of autonomous racing, model-based RL agents are still challenging to implement; however, Brunnbauer et al. [2021] introduced a model-based controller using a deep neural network that sometimes performs even better than model-free state-of-the-art methods. That algorithm is model-learned and uses a simple Follow The Gap approach to generate training data to

learn the model of the environment.

There are two main types of model-free algorithms: value-based and policy-based. The policy-based methods will focus on optimising policies; they can be either gradient-based or gradient-free. Gradient-based algorithms use gradient ascent to optimise the parameters of the policy. In contrast, gradient-free optimises a "surrogate" objective function, like the Proximal Policy Optimization (PPO) introduced in Schulman et al. [2017]. The value-based approach instead learns the state or the state-action value. Value-based methods can either evaluate and improve the same policy used for decision-making during learning (on-policy) or evaluate and improve a different policy than the one used for making actions. The most famous off-policy method is Q-Learning, which we will discuss in detail in Subsection 2.2.1. From Q-Learning were introduced Deep Q Networks (DQN), discussed in Subsection 2.2.2 and introduced in Mnih et al. [2015]. The Deep Deterministic Policy Gradient method, discussed in Subsection 2.2.3 was then introduced in Lillicrap et al. [2015] as a combination of policy and value-based approaches.

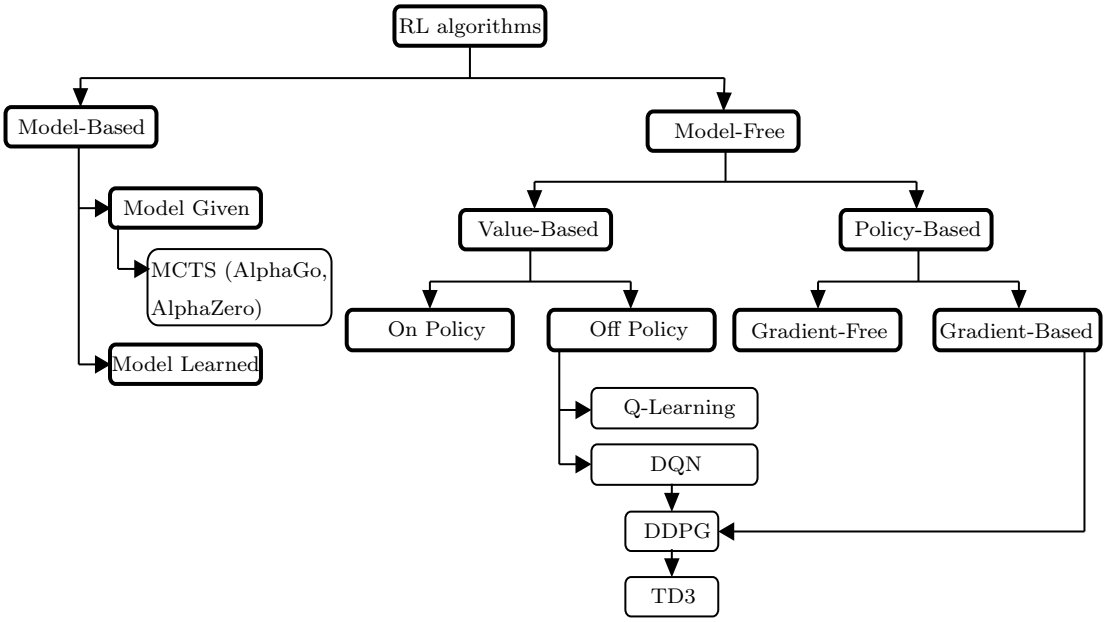

FIGURE 2.2: Reinforcement Learning Taxonomy

2.1.2 Markov Decision Process

A basic RL process can be modelled as a Markov Decision Process (MDP), as explained in Silver [2015] and Garcia and Rachelson [2013]. To understand how it works and can be applied to autonomous racing, we first need to define the Markov property: a state follows this property if the future is independent of the past given the present,

which means that the present contains all the information to determine the future. All MDPs are based on this assumption. MDPs involve policies which define how the agent behaves. As shown in Figure 2.3, they can be either deterministic ($a = \pi(s)$) or stochastic ($\pi(a \mid s) = \mathbb{P}(A = a \mid S = s)$. Stochastic policies are beneficial when the environment is uncertain: for autonomous racing, this is the case when other cars are taking part in the race. In the next parts, we will use both deterministic and stochastic policies, as well as quasi-deterministic policies like the ϵ-greedy one (Subsection 2.1.3).

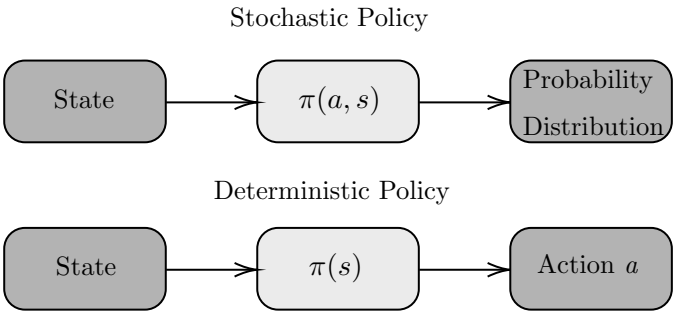

FIGURE 2.3: Comparison between deterministic and stochastic policies

Next, the agent's model is a prediction of what the environment will do next, with P the prediction of the next state and R of the next reward:

$$\begin{cases} P^a_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a] \\ R^a_s = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right] \end{cases} \tag{2.1}$$

In the specific case of autonomous racing, it is quite challenging to implement a model-based algorithm because the "rules" of a race are much more complicated to define than, for example, the rules of chess (Brunnbauer et al. [2021]): we will thus mostly work with model-free methods.

Those concepts can be combined into a Markov Reward Process (Figure 2.4). It consists of a tuple $\langle S, P, R, \gamma \rangle$ with:

- S a finite set of states
- P a state transition probability matrix ($P_{ss'} = \mathbb{P}\left[S_{t+1} = s' \mid S_t = s\right]$
- R a reward function, $R_s = \mathbb{E}\left[R_{t+1} \mid S_t = s\right]$ with $R : S \to \mathbb{R}$
- γ a discount factor, $\gamma \in [0, 1]$

We are here using a very simplified example of a racing car going through a corner: this will be used as a running example in all sections. The MRP is expressing the idea of delayed reward: if the agent accelerates, it will receive more short-term reward but will get highly penalised when hitting the wall; we will now introduce functions to "solve" this MRP by finding the path that gives the most reward. In this very simplified example, we consider that once the car has reached the state "accelerate again", it can't accelerate any more and has to go to the state "Hit wall".

The goal is to give a very general idea of how the RL controller of an autonomous racing car works; even though the actual algorithms we will implement on the F1Tenth platform are much more sophisticated, the idea is the same.

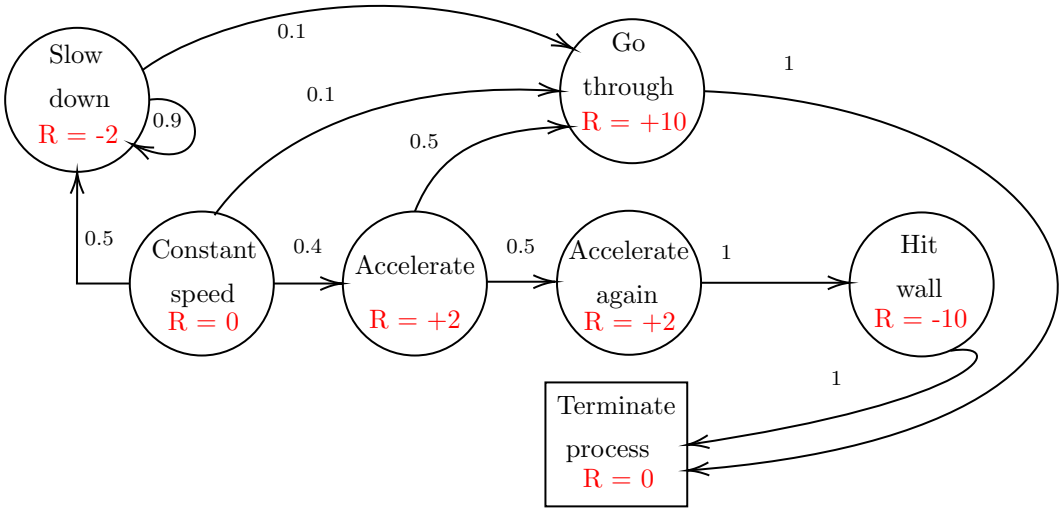

FIGURE 2.4: Markov Reward Process Example

A value function can then be defined as the total discounted reward of a policy starting in state s (Equation 2.2):

$$v_\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \mid S_t = s \right] \tag{2.2}$$

The discount factor γ represents the compromise between short and long-term decision-making: γ close to 0 will lead to a preference for short-term reward, whereas γ close to 1 will lead to the opposite. Most MDPs are discounted, partly because the discount conveniently avoids infinite return in cyclic MDPs, although it is possible to use *undiscounted* MDPs if all the sequences terminate. Furthermore, this is also closer to human and animal behaviour (preference for immediate reward), and it may be helpful if the

reward is financial (more interest earned with immediate reward).

We can then define the return G_t as the total discounted reward from the time-step t (Silver [2015]):

$$G_t = R_{t+1} + \gamma\, R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.3}$$

The value function of the MRP can then be defined as the expected return starting from state s:

$$\begin{aligned} v(s) &= \mathbb{E}\left[G_t \mid S_t = s\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma\, v(S_{t+1}) \mid S_t = s\right] \end{aligned} \tag{2.4}$$

A MDP can now be defined as a MRP with A added as a finite set of actions: $\langle S, A, P, R, \gamma \rangle$ (Garcia and Rachelson [2013]). We will define a policy π as a distribution over actions given states: $\pi(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$. The state-value function $v_\pi(s)$ of an MDP is then the expected return starting from s and following π. Similarly, the action-value function $q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$ is the expected return starting from s, doing a and following π. Those two functions can again be decomposed as the sum of the immediate reward and the discounted value of the successor state (Equations 2.5 2.6):

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma\, v_\pi(S_{t+1}) \mid S_t = s] \\ &= \sum_{a\ \in A} \pi(a \mid s)\, q_\pi(s, a) \\ &= \sum_{a \in A} \pi(a \mid s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \right) \end{aligned} \tag{2.5}$$

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma\, q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \\ &= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \\ &= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a' \mid s')\, q_\pi(s', a') \end{aligned} \tag{2.6}$$

Then, the optimal state-value function $v_*(s) = \max_\pi v_\pi(s)$ and the optimal action-value

function $q_*(s,a) = \max_\pi q_\pi(s,a)$ can be defined respectively as the maximum value function and the maximum action-value function over all policies. They are represented in the example MDP Figures 2.5 and 2.6.

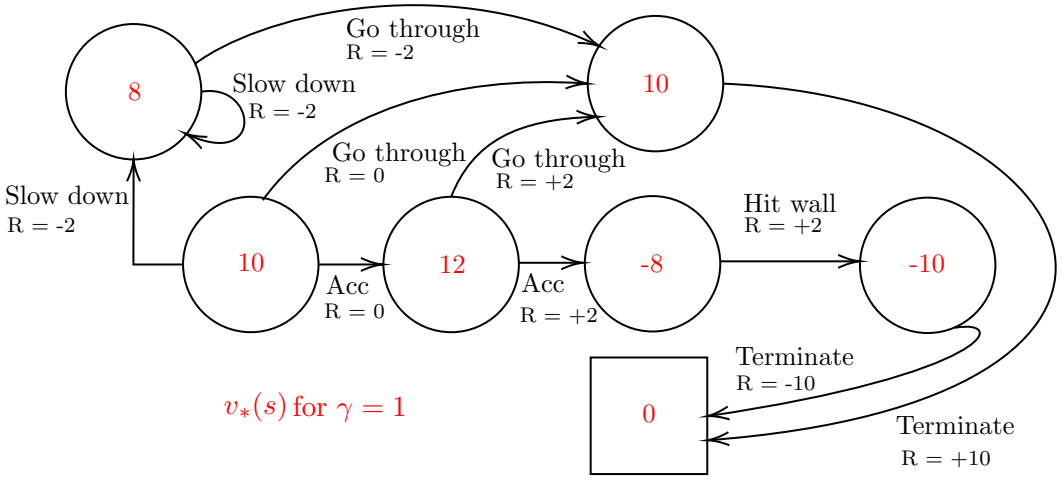

FIGURE 2.5: Optimal Value Function of an MDP

The values in red are $v_*(s)$ calculated with $\gamma = 1$. They represent the best value the agent can get from the MDP when starting in state s.

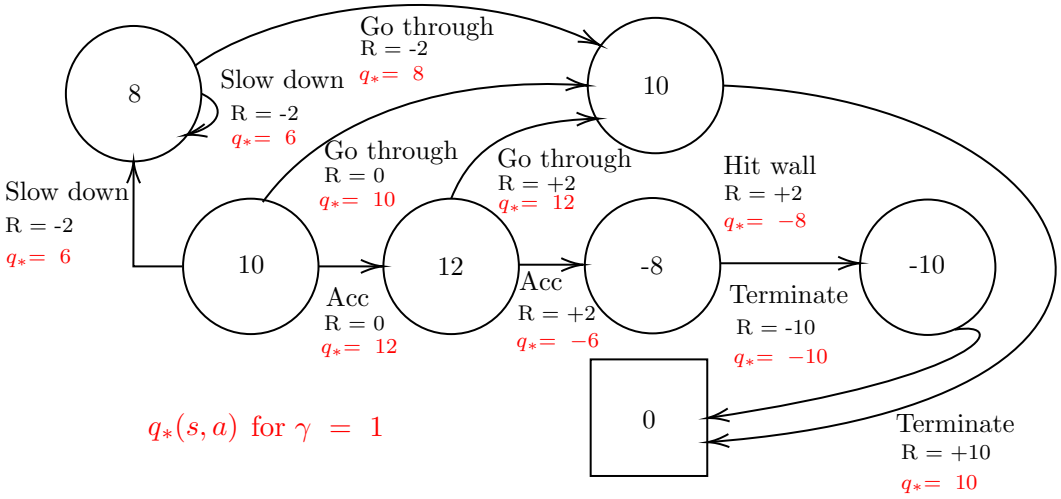

FIGURE 2.6: Optimal Action-Value Function of an MDP

The goal of an RL algorithm is to find the best approximate of $q_*(s,a)$ to solve the MDP. Once $q_*(s,a)$ is known, the optimal policy $\pi_* \geq \pi, \forall\pi$ ($\pi \geq \pi'$ if $v_\pi(s) \geq v_{\pi'}(s), \forall s$) can be found by maximising over $q_*(s,a)$, as can be seen in Figure 2.7.

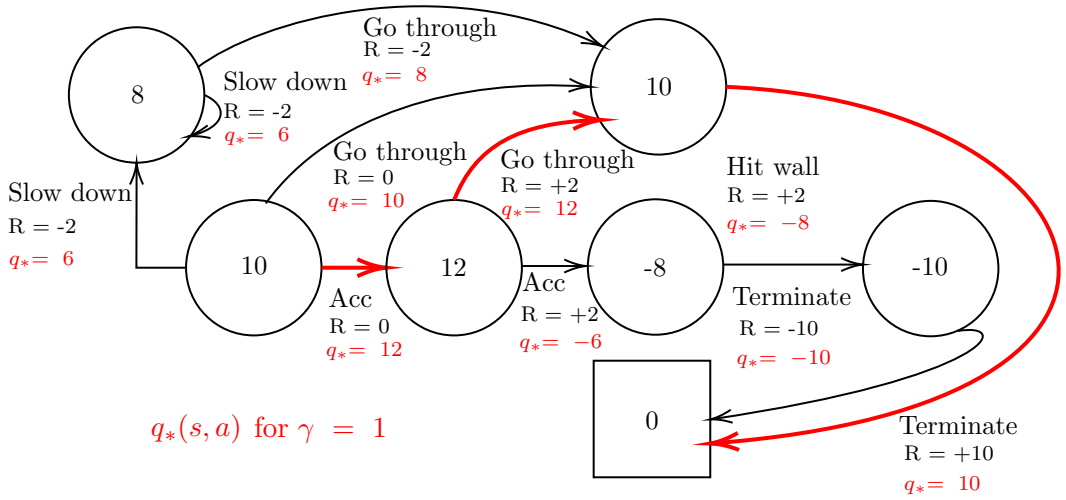

FIGURE 2.7: Optimal Policy of an MDP

As expected, the optimal policy for the car is to accelerate only once to go through the corner. Even though the example was trivial, the process is the same for all RL algorithms based on Q-Learning (Subsection 2.2.1): The agent moves along the edges with the highest q_*, it maximises over $q_*(s, a)$.

Finally, the Bellman equations for v_* and q_* can be expressed as such (Equation 2.7):

$$\begin{aligned} v_*(s) &= \max_a q_*(s, a) \\ &= \max_a R_s^a + \gamma \sum_{s \in S} P_{ss'}^a v_*(s') \\ q_*(s, a) &= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \\ &= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s', a') \end{aligned} \tag{2.7}$$

MDPs can be extended to Partially Observable MDPs (POMDPs) that are a generalisation of MDPs, where the agent's observation is a subset of its state Silver [2015]. This represents a situation where the agent indirectly observes the environment: it could be a robot with a limited field of view or a trader with only limited access to financial market data. POMDPs can thus be represented as a tuple $\langle S, A, O, P, R, \gamma, Z \rangle$ with O a set of observations and Z an observation function such as $Z_{s'o}^a = \mathbb{P}[O_{t+1} = o \mid S_{t+1} = s', A_t = a]$. The agent has its belief state $b(h)$, which is a probability distribution over all states

conditioned on the agent's history h (Equation 2.8):

$$b(h) = (\mathbb{P}[S_t = s^1 \mid H_t = h], ..., \mathbb{P}[S_t = s^n \mid H_t = h]) \tag{2.8}$$

POMDPs can then be reduced as a history tree storing the succession of alternating actions and observations and a belief tree storing the probability of being in a state conditioned on this succession of actions and observations.
In the case of autonomous racing with a model-based controller, the problem can be formalised as a POMDP Brunnbauer et al. [2021]. The deterministic reward function R is chosen to incentivise safe, smooth and fast control of the car, and the observation function Z would, in our case, output the equivalent of a LiDAR scan with a limited field of view.

2.1.3 Exploration vs. exploitation problem

One of the most fundamental questions of Reinforcement Learning is how an agent can compromise between exploration and exploitation. Exploitation consists of making the best decision given the current information (according to the agent's current value function). In contrast, exploration consists of doing something else to gather more information, which could lead to better rewards in the long run.
The best long-term strategy would be a compromise between exploiting and exploring to avoid getting stuck in local minima; this is the same dilemma someone could face when deciding between going to their favourite restaurant or trying out a new one. There are three main approaches for compromising between exploration and exploitation (Silver [2015]):

- Random exploration (e.g. the softmax or the ϵ-greedy algorithms, which introduces randomness into the action choice; this is the less sophisticated method.
- Optimism in the face of uncertainty: whenever the agent is uncertain about the value of an action, it should try it. The challenge is that the agent needs some way of measuring certainty; this is, however, a more "rigorous" approach than random exploration.

- The most systematic approach but also the most computationally difficult is to consider the agent's information itself as part of its state (the agent might be in a state where it has never tried a specific action).

There are two spaces in which an agent can explore: the state-action space (taking different actions) and the parameter space (trying out different policy parameters for a parametrised policy $\pi(A|S,u)$; the advantage of exploring the parameter space is that the exploration is more consistent than state-action exploration; however the downside is that the learning process may take more time to converge because the agent doesn't know if it has already made an action before and may thus do the same action expecting different results.

The exploration vs exploitation problem can be modelled as a multi-armed bandit process. A multi-armed bandit is a tuple $< A, R >$, with A the known set of actions ("arms") and R the unknown set of rewards. For each action there is an unknown reward defined by $R^a(r) = \mathbb{P}[R = r|A = a]$, and at each step t the agent chooses an action $A_t \in A$. The environment then generates a reward $R_t \sim R^{A_t}$, and the goal of the agent is to maximise the cumulative reward $\sum_{\tau=1}^{t} R_\tau$. The action-value function gives the mean reward for an action a, defined by: $q(a) = \mathbb{E}[R|A = A]$; the optimal value is then defined as $v_* = q(a^* = \underset{a \in A}{max}\, q(a)$. The regret can then be defined as the loss of opportunity for one step: $I_t = \mathbb{E}[v_* - q(A_t)]$; the total regret is then the total loss of opportunity over t steps: $L_t = \mathbb{E}\left[\sum_{\tau=1}^{t} v_* - q(A_\tau)\right]$. It should also be noted that maximising the cumulative reward is equivalent to minimising the total regret. The smaller the total regret, the better the policy is.

We can now define the count $N_t(a)$ as the number of times an action a has been selected, and the gap Δ_a as the difference in value between some action a and the optimal action a^*: $\Delta_a = v_* - q(a)$. Regret can finally be defined as a function of gaps and counts (Equation 2.9):

$$\begin{aligned} L_t &= \mathbb{E}\left[\sum_{\tau=1}^{t} v_* - q(A_\tau)\right] \\ &= \sum_{a \in A} \mathbb{E}[N_t(a)]\,(v_* - q(a)) \\ &= \sum_{a \in A} \mathbb{E}[N_t(a)]\,\Delta_a \end{aligned} \tag{2.9}$$

This definition can be understood as follows: we want to pick the actions ("arms") which are best as often as possible and the actions that are worst as infrequently as possible; the problem here is that the gaps are not known to the agent.

The ϵ-greedy algorithm that will be used in Algorithm 2 is a naive way of solving the exploration-exploitation problem: every time step there is a probability ϵ to select a random action and a probability $1-\epsilon$ to select $A = \underset{a \in A}{argmax}\, Q(a)$. Because ϵ is constant, it continues to explore the same way forever.

2.2 Reinforcement Learning applied to autonomous racing

2.2.1 Q-Learning

Algorithm 1 Q-Learning: Learn function $Q : X \times A \to \mathbb{R}$, Watkins [1989]

Require:
States $X = \{1, ..., n_x\}$
Actions $A = \{1, ..., n_a\}$ $A : X \Rightarrow A$
Reward function $R : X \times A \to \mathbb{R}$
Transition function $T : X \times A \to X$
Learning rate $\alpha \in [0, 1]$
Discount factor $\gamma \in [0, 1]$

Initialisation: $Q : X \times A \to \mathbb{R}$ with arbitrary values

while Q is not converged **do**
  Initialise s
  while s is not terminal **do**
    Choose a from s following a policy derived from Q (e.g.the ϵ-greedy policy)
    $r \leftarrow R(s, a)$
    $s^{'} \leftarrow T(s, a)$
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s^{'}, a^{'}) - Q(s, a) \right]$
    $s \leftarrow s^{'}$
  end while
end while
return Q

The Q-Learning algorithm introduced in Watkins [1989] is the most used model-free off-policy RL algorithm. With enough time and for any MDP, Q-Learning can determine the optimal action-selection policy, which was first rigorously proved in Watkins and Dayan [1992]. The algorithm is a function calculating the "quality" of a state-action combination: $Q : S \times A \to \mathbb{R}$. This scalar value is the Q-value associated with the state-action combination. The training process can be imagined as updating the Q-values stored in a 2D table of state-actions combinations. The core of the Q-Learning

algorithm is the Bellman equation, which updates the Q values. The new value of Q is the sum of three different quantities:

- $(1-\alpha)Q(s_t, a_t)$, which is the current Q-value weighted with the learning rate α
- $\alpha\, r_t$, which corresponds to the reward obtained if action a_t is taken when the agent is in state s_t, multiplied by the learning rate
- $\alpha\, \gamma \max_a Q(s_{t+1}, a)$, which is the maximum reward possible from state s_{t+1} multiplied by the learning rate and the discount factor

It is important to note that Q-Learning is off-policy because when updating the Q-value on line 7, it uses the Q-value of the following state s' and the greedy action a'. It doesn't matter that the policy followed to choose a from s on line 4 is not a greedy policy, Q-Learning will assume the agent is following a greedy policy nevertheless. This is similar to the Deep Deterministic Algorithm, which we will look at in section 2.2.3. The policy derived from Q used to choose a is sometimes an ϵ-greedy policy, which was introduced in Subsection 2.1.3.
It is important to consider the learning rate α, the discount factor γ and the initial conditions Q_0.

- With a learning rate of 0, the agent won't learn anything and will only exploit his prior knowledge. With $\alpha = 1$, it will only look at the most recent data. The learning rate can also be a function of time, decreasing when the training has progressed enough.
- γ represents the compromise between short ($\gamma \to 0$) and long ($\gamma \to 1$) sightedness. The discount factor is generally chosen close to 1, starting with a lower factor and increasing it over time to accelerate the training.
- The initial conditions Q_0 are also important. If they are not carefully chosen, they may lead to a situation where states that have already been visited may have a value superior to states that haven't yet. The agent would thus explore less at the beginning of the training and get stuck.

2.2.2 Deep Q-Learning

Deep Q-Learning is an improvement over Q-Learning introduced by DeepMind in Mnih et al. [2015]. Their goal was to develop an algorithm that could solve a wide range of challenging tasks that cannot be handled with Q-Learning. The process combined RL with deep CNN to deal with never encountered states. The NN approximates the optimal action-value function representing the maximum sum of rewards r_t with discount γ achievable by a policy $\pi = P(a|s)$ after making an observation s and an action a (Equation 2.10):

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi\right] \tag{2.10}$$

The algorithm has the following structure (Algorithm 2):

Algorithm 2 Deep Q-Learning with experience replay, Mnih et al. [2015]

Initialisation:

Initialise replay memory R to capacity N

Initialise action-value function $Q(s, a, \theta)$ with random weights θ

Initialise target action-value function Q_{target} with weights $\theta^{'} = \theta$

1: **for** episode = 1, episode $\leq M$ **do**

2: Initialise state s_t

3: **for** $t = 1, t \ \leq T$ **do**

4: Choose a_t from s_t following the ϵ-greedy policy

5: Execute action a_t and observe reward r_t and new state s_{t+1}

6: Store transition (s_t, a_t, r_t, s_{t+1}) in R

7: Set $s_{t+1} = s_t$

8: Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R

9:

$$\text{Set } y_i = \begin{cases} r_i & \text{for terminal state } s_{t+1} \\ r_i + \gamma \max\limits_{a^{'}} Q_{target}\left(s_{t+1}, a^{'}, \theta\right) & \text{for non-terminal state } s_{t+1} \end{cases}$$

10: Perform a gradient descent step on $\left(y_i - Q\Big(s_t, a_i, \theta\Big)\right)^2$ with respect to θ

11: Every C steps reset $Q_{target} = Q$

12: **end for**

13: **end for**

This algorithm differs from standard online Q-Learning in two ways. Firstly, it uses experience replay: the replay buffer stores the agent's experiences at each time-step (line 6) over many episodes. Inside the inner for-loop, the Q function is updated using a subset of experiences randomly sampled from the replay buffer. There are several advantages over online Q-Learning: an experience can be used several times for weight updates, which allows for greater data efficiency; it avoids overfitting by breaking the correlation between consecutively sampled experiences, and it helps avoid unwanted feedback loops that could stick the agent in a local minimum (Tsitsiklis and Van Roy [1997]). Thus experience replay helps average the agent's behaviour by avoiding oscillations and divergences in θ.

The second difference with online Q-Learning is that the algorithm uses a different network Q_{target} to generate the targets y_i (line 9). Every C step Q_{target} is updated to Q, which creates a delay between the time Q is updated and the time the update affects the targets y_i. Again, this is to avoid having diverging or oscillating parameters. As described in Mnih et al. [2015], DeepMind reached super-human performance levels on several Atari games.

2.2.3 Deep Deterministic Policy Gradient method

The Deep Deterministic Policy Gradient algorithm is a mode-free, off-policy algorithm, first introduced in Lillicrap et al. [2015]. It combines Deterministic Policy Gradients and Deep Q-Learning to learn a policy. As shown in Figure 2.3 Deterministic is opposed to stochastic: the policy takes a state as input and returns a single action; there are no probabilities involved.

DDPG is thus both value-based (approximating the Q-function) and policy-based (using the Q-function to learn the policy): this can be seen in Figure 2.2. This approach is quite similar to Q-Learning: if the agents knows the optimal action-value function $Q^*(s,a)$, it can then find the optimal action $a^*(s)$ by solving: $a^*(s) = arg\ \underset{a}{max}\ Q^*(s,a)$. The advantage of DDPG over Q-Learning is that it is adapted for continuous action spaces. This is related to the way $\underset{a}{max}\, Q^*(s,a)$ is computed: when there is only a finite number of actions, it is straightforward to compute by calculating the Q-values corresponding to all possible actions and selecting the best. In a continuous action space, we can consider the $Q^*(s,a)$ to be differentiable with respect to a, which lets us define $\mu(s)$, a rule to learn a policy. This way, $max_a Q(s,a)$ can be approximated as $max_a Q(s,a) \approx Q(s,\mu(s))$.

We will now look at the equations involved in the part of DDPG that is learning the Q-function. The Bellman equation giving the optimal action-value function Q^* is defined by Equation 2.11:

$$Q^*(s,a) = \underset{s' \sim P}{E}\left[r(s,a) + \gamma\ \underset{a'}{max}\ Q^*(s',a')\right] \tag{2.11}$$

Here, $s' \sim P$ means that the next state s' is sampled by the environment from a probability distribution P. The Bellman equation is then used to learn the approximation of Q^*. There are three main elements (Figure 2.8):

1. the actor network taking a state s as input and returning an action a

2. the critic network taking as input a state a and an action a and returning the corresponding $Q_{value} = r + \gamma\, Q_{next}$

3. the memory or replay buffer storing a subset of the past experiences of the agent: the state s, action a, reward associated r and the next state s.It shouldn't contain too many previous experiences as that would make the training very slow but should be large enough to avoid overfitting.

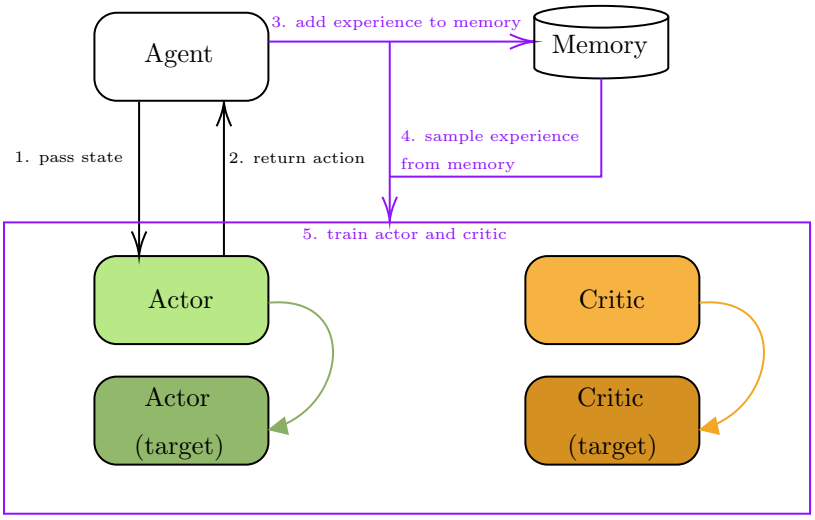

FIGURE 2.8: Deep Deterministic Policy Algorithm Process

It is important to note here that, like Q-Learning, DDPG is an off-policy algorithm which means that the learning proceeds not from actions taken using the current policy $\pi(a|s)$ but from possibly outdated policies from the replay buffer. This is possible because, as demonstrated in Watkins [1989], the Q-value of the following state $s^{'}$ contained in the Bellman equation is obtained following the greedy action $a^{'}$; it estimates the return for a state-action pair assuming a greedy policy is followed. This is why the Bellman equation is not concerned about the current policy and thus why DDPG is an off-policy algorithm. It is also important to note that because DDPG is off-policy, it should be more stable than on-policy methods, although it should converge slower.
The detailed training process is presented in Figure 2.9:

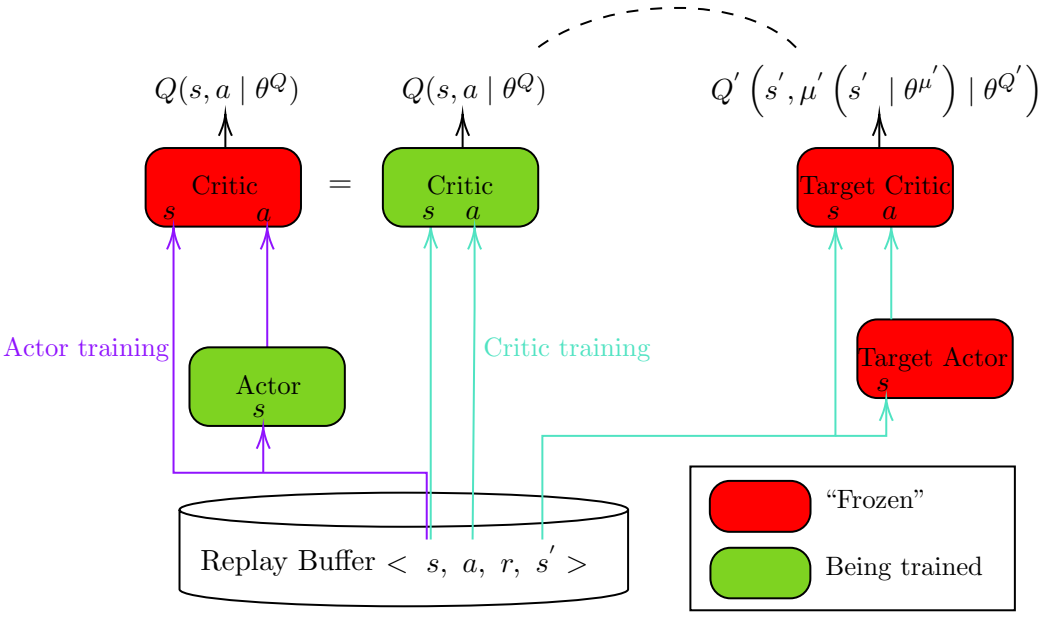

FIGURE 2.9: Detailed training process of the DDPG algorithm

We will now consider the detailed training process (Figure 2.9). We will look first at the training of the actor network. Assuming we have the critic network already trained, the observation from the replay buffer is fed into the actor and the critic, and the output action of the actor is fed into the critic network. The goal is then to maximise the Q-value output of the critic network.

We have to use a copy of the critic and the actor networks to train the critic network. We feed into the actor and the critic the following observation, and the output of the actor network, which is the following action, is fed into the critic network. The critic outputs Q_{next}; this Q_{next} is then compared to the Q value output of the target critic network fed by the observation and action of the replay buffer. The goal here is to minimise $|Q - (r + \gamma \cdot Q_{next})|$, with the discount factor γ a hyper-parameter of the algorithm and the reward r stored in the replay buffer (and corresponding to the action a). This is the part where the target critic and actor are necessary because without them; the problem would be that the critic and the actor would depend on the same parameters (θ^Q for the critic and θ^μ for the actor) that we are trying to train, which would make the algorithm unstable. The target networks with parameters $\theta^{Q'}$ for the critic and $\theta^{\mu'}$ for the actor are lagging the actor and critic networks by one time-step and are updated as such (Equation 2.12):

$$\theta' \leftarrow \rho\, \theta' + (1 - \rho)\theta \tag{2.12}$$

with ρ a hyper-parameter between 0 and 1. The whole training process is summarised in Algorithm 3.

For the policy training part (the training of the actor network), we want to learn a deterministic policy μ_θ which will maximise $Q(s, a \mid \theta^\mu)$. For the Q-Learning part (the training of the critic network), we want to minimise the difference in Equation 2.13:

$$\left(r + \gamma\, Q^{'}(s, \mu^{'}\left(s \mid \theta^{\mu^{'}}\right) \mid \theta^{Q^{'}}\right) - Q\left(s, a \mid \theta^{Q}\right) \tag{2.13}$$

Assuming we are approximating using a NN $Q(s, a \mid \theta^Q)$ with θ^Q the parameters and that we are storing in a replay buffer R the following variables: $\left\langle s, a, r, s^{'}\right\rangle$, we can set-up a mean-squared Bellman error function (Equation 2.14), with the goal of minimising it:

$$L(\theta, R) = \underset{(s,a,r,s^{'})\sim R}{\mathbb{E}}\left[\left(r + \gamma Q^{'}\left(s^{'}, \mu^{'}\left(s^{'} \mid \theta^{\mu^{'}}\right) \mid \theta^{Q^{'}}\right) - Q(s, a \mid \theta^{Q}\right)^2\right] \tag{2.14}$$

Algorithm 3 Deep Deterministic Policy Gradient, Lillicrap et al. [2015]

Input: random initial weights θ^Q for critic network $Q(s, a|\theta^Q)$ and θ^μ for actor network $\mu(s|\theta^\mu)$

1: Initialise target network $Q^{'}$ and $\mu^{'}$ with weights $\theta^{Q^{'}} \leftarrow \theta^Q$, $\theta^{\mu^{'}} \leftarrow \theta^\mu$

2: Initialise empty replay buffer R Initialise a random process N for action exploration

3: Receive initial observation state s_1

4: **for** $t = 1, t \leq T$ **do**

5: Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to current policy and exploration noise

6: Execute action a_t and observe reward r_t and new state s_{t+1}

7: Store transition (s_t, a_t, r_t, s_{t+1}) in R

8: Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R

9: Set $y_i = r_i + \gamma Q^{'}\left(s_{i+1}, \mu^{'}\left(s_{i+1} \mid \theta^{\mu^{'}}\right) \mid \theta^{Q^{'}}\right)$

10: Update critic by minimising the loss: $L = \frac{1}{N}\sum_i \left(y_i - Q(s_i, a_i \mid \theta^Q)\right)^2$

11: Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q\left(s, a \mid \theta^Q\right)|_{s=s_i, a=\mu(s_i)}\, \nabla_{\theta^\mu}\mu(s \mid \theta^\mu)\ |_{s_i}$$

12: Update the target networks:

$$\theta^{Q^{'}} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q^{'}}$$

$$\theta^{\mu^{'}} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu^{'}}$$

13: **end for**

Line 10, the loss is given by the average of the squared differences between the target

action-value y_i and the expected action-value $Q\left(s_i, a_i \mid \theta^Q\right)$, with the expected action-value being given by the critic network Q. Line 9, the target action-value is calculated as the sum of the reward r_i and the discounted action-value where the target critic network $Q^{'}$ takes the state s_{i+1} and the action returned by the target actor $\mu^{'}$, mapping the state s_{i+1} to the action $\mu^{'}\left(s_{i+1} \mid \theta^{\mu^{'}}\right)$.

Then line 11 the actor is updated with the sampled policy gradient, which is the average of the action values given by the critic Q with parameters θ^Q that takes the state s and the action a as input, where the action

Finally, line 12, the algorithm uses the coefficient τ, $\tau \ll 1$ to make the weights $\theta^{Q^{'}}$ and $\theta^{\mu^{'}}$ change slowly instead of directly copying θ^Q and θ^μ, which improves the stability of the process.

2.3 Neural Networks

2.3.1 Multi-Layer Perceptron

The Multi-Layer Perceptron (MLP) is a type of NN composed of perceptrons. The advantage over a simple perceptron is that a perceptron is a linear classifier, whereas an MLP is non-linear and can classify classes that are not linearly separable. The perceptrons are organised in at least three layers: the input layer, one or more hidden layers and the output layer. In the example Figure 2.10, each connection between nodes is represented in a shade of grey depending on the value of the associated weight.

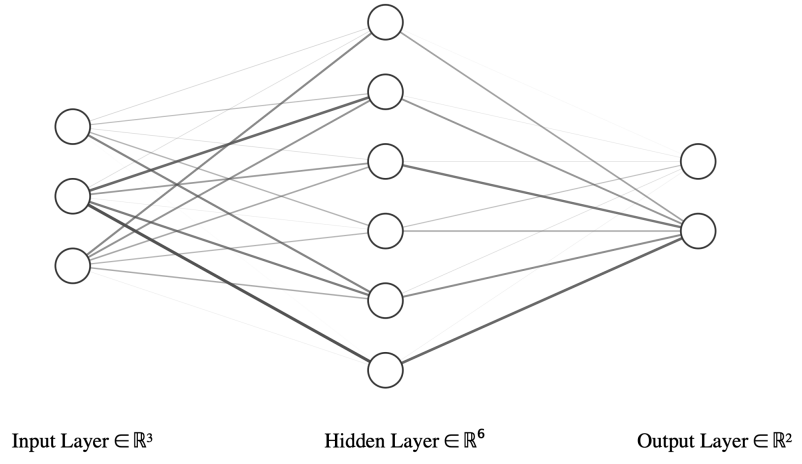

FIGURE 2.10: Representation of a simple MLP

An MLP is a function of several parameters and hyper-parameters. The parameters of the MLP are the weights θ of each connection, and the hyper-parameters are:

- The number of hidden layers
- The number of neurons per layer
- The weights initialisation values
- The activation functions of each layer
- The learning rate α
- The number of epochs N
- The loss function for backpropagation

The training process is relatively straightforward and consists of tuning the values of the weights for N epochs (Popescu et al. [2009]). Each epoch goes through several cycles of forward and back propagations over a subset of the training set: the network output is calculated using the weights and activation functions (forward propagation). This output is compared to the expected output using the loss function. The weight values are then updated using the loss function, starting from the output layer and moving back toward the input layer (backpropagation).

The Deep Q-Learning (Subsection 2.2.2) and the DDPG (Subsection 2.2.3) algorithms that we introduced previously are implemented using NNs. In the case of autonomous racing using LiDAR data, the input layer could have as many neurons as there are LiDAR points. The output layer would only have two neurons: one returning the steering angle (maybe a value $\theta \in [-\pi, \pi]$, with θ in radians) and the other returning the motors command $m \in [0, 1]$ for example.

2.3.2 Convolutional Neural Networks

Convolutional Neural Networks are a type of NNs better suited to handle 2 or 3-dimensional data (O'Shea and Nash [2015]). The problem with using simple MLP for such data is that the NN loses all the spatial structure of the image with $h \times w$ pixels into a 1D array of $h \times w$ values. CNNs can preserve the spatial relationship between

the pixels and let the NN learn the internal features of the data by using three main types of layers: convolutional, pooling and fully connected layers; an example is given in Figure 2.12.

The convolutional layer is defined by the filters, the feature maps, the stride and the padding. The filters correspond to the "neurons": they have input weights and output a value. For a 2D image, the filter is a square of a specific size (Figure 2.11).

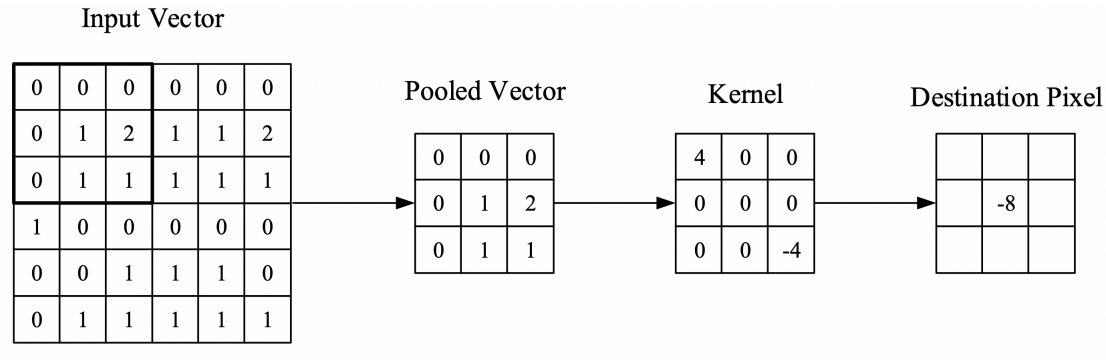

FIGURE 2.11: Example of a Convolutional Neural Network, taken from O'Shea and Nash [2015]

The feature map is the output of a filter applied to the previous layer of the network. When the filter is moved across the input data, it returns different values that are added to the feature map; the amount by which the filter is moved is the stride, and padding can be added to the border of the input if the filter doesn't perfectly "line-up" with the input. The size of the feature map is a function of the input size (height, width and depth) v, the size of the filter f, the amount of padding added p and the stride s such as (Equation 2.15):

$$\text{Output size} = \frac{(v - s) + 2p}{s + 1} \tag{2.15}$$

The convolutional layer is thus a great way of reducing the input size while preserving the spatial relationship.

The pooling layer aims to reduce the dimensionality of the data slowly. The filter of size f moves across the input data with a stride s and outputs the max value (for max-pooling) of its area $s \times s$. It thus scales down the size of the input data into something more manageable. Because data is lost in the process, the size of the filter should be kept to 2×2 or 3×3.

Finally, the fully connected layer of a CNN corresponds to an MLP as defined in Subsection 2.3.1.

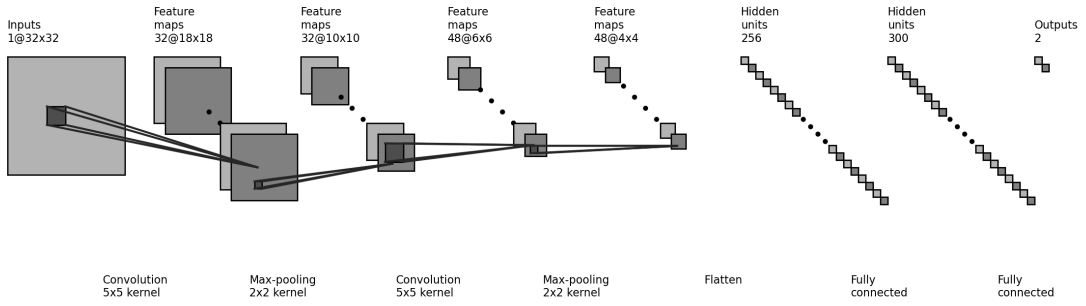

FIGURE 2.12: Example of a Convolutional Neural Network, made using http://alexlenail.me/NN-SVG/LeNet.html

In this example, the 2D input data goes through a max-pooling layer, a convolutional layer, another max-pooling layer, then gets flattened and goes through a fully connected network with an input layer of size 256, a hidden layer of size 300 and an output layer of size 2 (corresponding to steering and motors signals).

In the case of using LiDAR data for a racing car controller, CNNs are shown to generally offer better results (Bosello et al. [2022]): we will try to replicate those results and confirm that CNNs are superior when used with the controllers we will implement. We will implement fully connected NNs and CNNs to parse the LiDAR data feed after some pre-processing; more information about technical details is given in Chapter 3.

2.4 Robot Operating System and F1Tenth System

2.4.1 ROS framework

Robotics has long been a playground for Reinforcement Learning algorithms (Sutton and Barto [2018]), and many implementations of those algorithms have been done using Robot Operating System (ROS). ROS is a robotic software platform that is open-source (BSD license) and provides a vast amount of services, tools and libraries to serve as the framework of any robotic platform. Because ROS is open-source, it is straightforward to contribute to the project, and many users have developed and shared their own libraries.

ROS is also a meta-operating system, meaning it behaves like an operating system but runs on a Unix-based OS. The goal of ROS is to enable code sharing and collaboration among coders: ROS was thus designed as a distributed framework of processes called nodes that can be grouped in packages and stacks to be shared easily. A package is the smallest self-functioning unit that exists in ROS. Most ROS projects contain several packages that build up on each other and are declared as *dependancies* of each other. Packages are located inside a catkin workspace, built using the catkin build system with the command `catkin_make`. Catkin is the official build system of ROS and was designed to replace rosbuild. A catkin workspace containing packages has the following general structure (Figure 2.13):

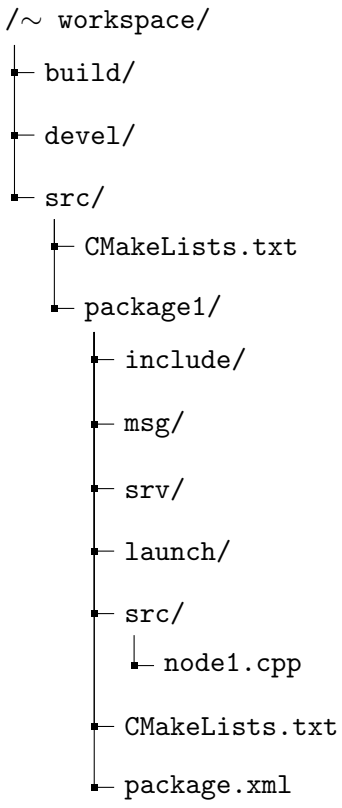

FIGURE 2.13: Directory structure of a ROS workspace

Each package has to have the `CMakeList.txt` and the `package.txt` files. The `CMakeList.txt` tells catkin how to compile the package, with the locations of the different libraries and subdirectories. The `package.xml` contains several elements, including the dependencies (other packages) used by the package, the name and a description of the package, the tools used to build the package and nodelets, which are external tools for the package. The package architecture is, however, not a rigid as the workspace architecture, the

source folder can be renamed, and some nodes can be put in other packages.
The architecture of a ROS package can be represented as a graph, with nodes connected by edges that are called *topics* (Figure 2.14). Each node can send and receive messages to other nodes using the topics. The primary process in this graph is called the *ROS Master*. It registers all the nodes and sets up the node-to-node communications for each topic. A ROS program is decentralised because the messages and service calls don't go through the ROS Master; they are all peer-to-peer communications. This is a very modular architecture that works very well when robots themselves are made of several modules communicating with each other.

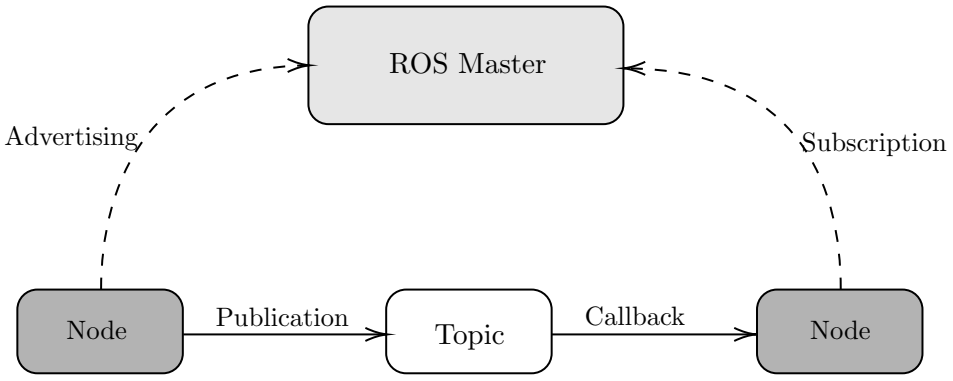

FIGURE 2.14: ROS nodes interactions

A ROS node is a process running as part of the ROS graph, written either in Python or C++. They all have a name that must be registered with the ROS Master and represent most of the code written for a ROS project. To send a message to a topic, a node has to register as a publisher; to receive messages, it has to register as a subscriber. The messages themselves can be any type of data: in the case of the F1Tenth system, they will contain sensor data with the LiDAR points, the actuators commands for the steering and the motor control commands for the speed.

2.4.2 F1Tenth framework

The F1Tenth platform was designed as a teaching and open-source research tool and, as such, is perfect for this project. It consists of a small racing car composed of a lower and upper chassis with a 5000 mAh battery and several autonomy elements: a WiFi antenna, a UST-10LX LiDAR, an NVIDIA Jetson as the Electronic Control Unit (ECU), an NVIDIA Jetson NX GPU for parallel computing, a speed controller and a power

board.

The main interest of the F1Tenth car is that it offers hardware and software similar to full-scale racing systems: realistic dynamics, with Ackermann steering and high maximum speed. The idea of Ackermann steering is simple: in a corner, the inner front wheel has a larger turning angle than the outer front wheel so that the tires are not slipping sideways.

The F1Tenth simulator used for this project is the default one: developed as a lightweight 2D simulator built using ROS, it is written in C++ and displays 3D graphics using RViz. RViz is a three-dimensional visualiser used to see how a robot evolves in its environment. Like most ROS tools, it is highly modular, with several different types of visualisations. The F1Tenth simulator simulates the car motion and collisions, handles the publishing of LiDAR scans and odometry data and is built for the fast prototyping of racing algorithms, which corresponds perfectly to this MSc project. The basic ROS architecture of the simulator is presented in Figure 2.15. Another possibility would have been to develop our own simulator, but this would have been too time-consuming. The default simulator is good enough for our needs; however, it does not support more than one car; this could be a problem later if we want to test our algorithms in a multi-car racing environment.

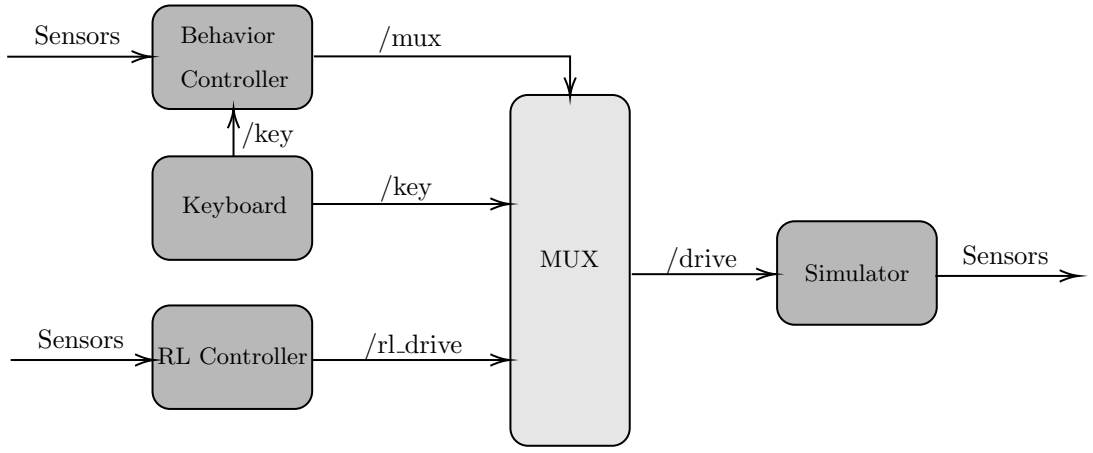

FIGURE 2.15: F1Tenth system architecture

The simulator was developed to be interchangeable with the actual car system. Once the training of the RL controller has been finished in the simulator, the simulator node can be replaced by the F1Tenth car itself, and if the topics' names are not changed, then the same code can be used to drive the car to evaluate the sim-2-real gap. This way, we can easily swap between different types of RL controllers, both in the simulator

and on the actual car. The `MUX` node listens to the `/mux` topic to know which controller is on; in our case, it would be the `RL Controller` one, so `MUX` listens to `rl_drive` and then transmits the message to `/drive`. Then either the `Simulator` node or the car itself listens to the `/drive` topic, and the state of the car gets updated. In the simulator, once a training epoch is completed, or once the car hits a wall, we can move it back to the start of the race track by publishing a Pose message to the `/pose` topic.

2.4.3 Wall Following controller for the F1Tenth platform

During the project, we will implement a Wall Following controller to be compared against RL methods. Unlike the RL controllers introduced in the previous sections, the Wall Following controller is not an ML algorithm; it relies on a Proportional-Integral-Derivative controller to maintain the desired distance with one of the walls of the race track. The challenge with this method is to measure the distance to the wall and anticipate the car's trajectory. To determine the distance of the car to the wall, 2 LiDAR measurements, l_1 and l_2 are made (Figure 2.16). The first measurement is perpendicular to the car's direction (angle of 0°), the second at an angle $0 < \theta < 90$. Using trigonometry, the angle α between the direction of the car and the wall can be expressed as Equation 2.16:

$$\alpha = \arctan\left(\frac{l_1 \cos(\theta) - l_2}{l_1 \sin(\theta)}\right) \tag{2.16}$$

D_t can then be expressed as $D_t = l_2 \cos(\alpha)$.

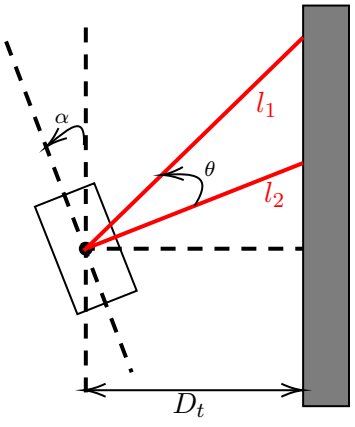

FIGURE 2.16: Wall Following geometry

However, as explained in Mangharam [2020], if we use the car's current position, the controller may not be able to respond fast enough to account for the computational delay, and the car's trajectory may not be optimal. Instead, the car's position at time step $t+1$ is used (Equation 2.17), with D_{t+1} the car's position at time step $t+1$, Δt the duration of a time step and v_t the speed of the car at time step t.

$$D_{t+1} = D_t + v_t \, \Delta t \, \sin(\alpha) \tag{2.17}$$

The Wall Following method relies on a Proportional-Integral-Derivative controller (Rivera et al. [1986]). A PID is a control loop method that applies accurate and optimal control to a system. A PID consists of a feedback loop: the error at time step t called $e(t)$ is the difference between the desired value and the feedback; the controller applies a correction based on the error itself, the integral of the error and the derivative of the error. A PID is much more accurate than a simple proportional controller: the integral term helps eliminate the residual error, and the derivative term helps react more quickly by "anticipating" the future values of the error.

The general equation of a PID controller is introduced in Equation 2.18. In our case, $s(t)$ is the steering angle, and $e(t) = D_{ref} - D_{t+1}$ is the difference between the desired distance between the car and the wall D_{ref} and the actual distance at time $t+1$. The constants K_p, K_i and K_d define how much each component of the PID contributes to the output $s(t)$.

$$s(t) = K_p \, e(t) + K_i \int_0^t e(t') \, dt' + K_d \, \frac{d}{dt}(e(t)) \tag{2.18}$$

The values K_p, K_i and K_d have to be tuned depending on the system. There are several methods to adjust the gains of a PID controller to achieve optimal control; in our case, we will start by changing the values manually, and if necessary, we will use some ROS packages or MATLAB tools.

Once the steering angle $s(t)$ is computed, the car's speed v_t could be adjusted: for example, if the steering angle is above a specific value, the car speed could be reduced to avoid losing grip on the track.

2.5 Summary

Through this literature review, we have introduced three main approaches to implementing an autonomous car controller, as well as the background necessary to understand them. We have looked at PID-based methods, such as wall following, policy gradient-based methods such as DDPG and off-policy value-based methods such as DQN.
We can now make a preliminary comparison of their main characteristics (Table 2.1) based on the authors' assessment of their work, looking particularly at four criteria:

1. The controller's reliability over multiple laps on the training race track.
2. The training time required to achieve optimal results (in steps).
3. The ability to overcome the Sim2Real gap; this will be investigated formally through Hypothesis 4 introduced in Section 3.2.
4. The ability to generalise to new race tracks; this will be investigated formally through Hypothesis 3 introduced in Section 3.2.

Paper	Reliability	Training time	Sim2Real gap	Generalisation
"Dreamer" from Brunnbauer et al. [2021]	Average	2 million steps	Minor	Very good
PID based wall following from Mangharam [2020]	Poor	N/A	Minor	Poor
Value-based DQN with NN from Bosello et al. [2022]	Average	550 000 steps	Average	Good
Value-based DQN using CNNs from Bosello et al. [2022]	Good	500 000 steps	Minor	Good
Policy gradient-based DDPG from Ivanov et al. [2020]	Good	1 million steps	Very minor	Good

TABLE 2.1: Preliminary comparison of controllers introduced in the literature review

Chapter 3

Requirements and Methodology

3.1 Deliverables

The workload for this MSc project is divided into three different deliverables:

1. This research report for the F21RP course (due April 20th).
2. The MSc thesis report (due August 15th). The report will test the hypotheses listed in Section 3.2 and draw conclusions from experimental results.
3. A software package providing implementation of three RL controllers for the F1Tenth platform (due August 15th), detailed in Section 3.4.

3.2 Research Hypotheses

I will start by implementing the Wall Following, DQN and DDPG controllers in the F1Tenth simulator. The DQN and DDPG controllers will then be tested on the training race track (the "RedBull Ring" track from Bosello et al. [2022]), using different sets of hyper-parameters to answer the first hypothesis. Because of a lack of time, the only hyper-parameter that was studied was the reward function; looking at the other hyper-parameters was left as future work. Also because of a lack of time, the DDPG controller wasn't implemented as explained in Chapter 6, and the remaining controllers were judged sufficient to test Hypothesis n°1.

Hypothesis n°1: *Deep Reinforcement Learning provides a real advantage, quantified by reliable metrics, for controlling an F1Tenth car over human control and PID-based methods like wall following. (Objectives 1,2,3)*

To verify the second hypothesis, the performance of the controllers will be compared on the "RedBull Ring" race track when using a fully connected NN architecture and a CNN. This will be performed in the simulator.

Hypothesis n°2: *CNNs offer better results over selected metrics than NNs for autonomous racing on the F1Tenth platform using data from a UST-10LX LiDAR. (Objectives 1,2,3)*

To verify the third hypothesis, the three controllers implemented will then be tested on two other race tracks presenting different challenges, namely the "Silverstone Circuit" and the "Circuit de Monaco" (both from Bosello et al. [2022]). Those tracks are introduced in Appendix A; importing race tracks has already been tried in the F1Tenth simulator, as explained in Section 3.5. DRL controllers were also trained on the Monaco track to compare their performance with those trained on the RedBull track.

Hypothesis n°3: *DRL-based controllers can generalise to new race tracks without affecting the metrics too much. (Objective 3)*

With the last hypothesis, I want to evaluate the impact of the gap between the F1Tenth simulator and the real world; developing controllers in the simulator would be useless if they can't be applied to the real world because the gap is too wide. Replicating the complete three circuits in the real world may prove too challenging; in the end, we decided to add a fourth track with an oval shape easier to reproduce in real life, as explained in Chapter 6.

Hypothesis n°4: *The sim2Real gap is negligible (minor changes of the metrics) and doesn't affect the performance of the controllers implemented. (Objective 5)*

3.3 Requirements Analysis

The requirements for the project have been divided into experimental requirements (regarding the research hypotheses) presented in Table 3.1, and software requirements presented in Table **??**.

3.3.1 Experimental Requirements

As will be explained in further detail in Chapter 6, because of a lack of time, the experimental requirements (highlighted in Table 3.1) related to the DDPG implementation were dropped, namely requirements E6 and E7.

3.4 Tooling and Implementation

A Python codebase will be used to implement this project. The code will be hosted on a public GitHub repository at the URL: `https://github.com/HL-Boisvert/MSc_Project_HW`.

The three controllers implemented (Wall Following controller, DQN controller, and DDPG controller) will be available on this GitHub repository, along with their documentation following requirement N1.

The Wall Following controller will be implemented in ROS first and will consist of a single .py file with the following methods and parameters (Figure 3.1):

FIGURE 3.1: Wall Following controller structure

When launching the .py file, the user will have the possibility to specify the K_p, K_i and K_d parameters.

Next, the DQN controller will be implemented using a more sophisticated architecture (Figure 3.2).

Ref	Name	Description	Obj	Priority	State
E1	Define a reward function	Define a reward function incentivising safe, smooth and fast car control.	1	High	Fully implemented
E2	Implement Wall Following	Implement the Wall Following controller in the simulator, with proper tuning of parameters k_p, k_d and k_i.	2	High	Fully Implemented
E3	Wall Following assessment	Assess the Wall Following controller in the simulator on all three maps (see Appendix A) against selected metrics.	3	High	Fully implemented
E4	Implement and train DQN	Implement and train the DQN controller in the simulator with several hyperparameters and NN architectures.	2	High	Fully implemented
E5	DQN assessment	Assess the DQN controller in the simulator on all maps against metrics.	3	High	Fully implemented
E6	Implement and train DDPG	Implement and train the DDPG controller with several hyperparameters and NN architectures.	2	Medium	Not implemented
E7	DDPG assessment	Assess DDPG in the simulator on maps against metrics.	3	Medium	Not implemented
E8	Sim2Real gap assessment	Assess all methods on recreated portions of the 3 maps against metrics; drive the car manually to compare metrics.	4	Medium	Partially implemented
E9	Overall Conclusions	Answer all hypotheses.	5	High	Fully implemented

Table 3.1: Experimental Requirements

Reference	Name	Description	Priority	Status
F1	Compatibility with F1Tenth platform	The controllers should be easily implementable on the F1Tenth platform	High	Fully implemented
F2	Error Handling	Proper handling of errors in Python	Medium	Partially implemented
F3	Easy parameter change	Let the user specify parameters when launching the .py file; parameters depend on the algorithm chosen.	Low	Fully implemented
F4	Training progress visualisation	Show the user the evolution of the average cumulative reward during the training.	Low	Not implemented
F5	Logging	For DRL algorithms, states, rewards actions and states should be written in a log file for debugging	Low	Partially implemented
F6	Implementation with other autonomous car platforms	Basic instructions should be given to implement the controllers on platforms others than F1Tenth	Low	Not implemented

TABLE 3.2: Software Requirements

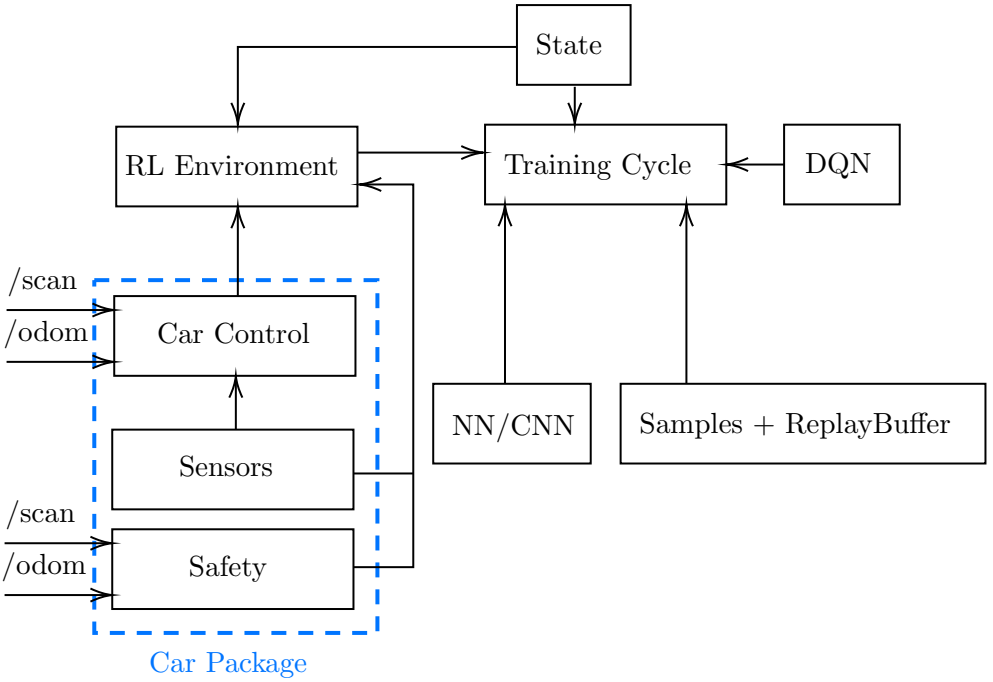

FIGURE 3.2: DQN controller structure

The program would be launched in the simulator using the following commands:

```
$ source devel/setup.bash
$ roslaunch f1tenth_simulator simulator.launch
$ python3 training_cycle.py --simulator
```

The user has the possibility to specify hyper-parameters values (discount, learning rate, maximum number of steps, size of the minibatch sample, NN architecture) when launching the program using:

```
$ python3 training_cycle.py --simulator --parameter1=value1 ...
```

Similar commands are used to launch the controller on the physical car.

The DDPG controller will be implemented last, using a similar architecture to the DQN but adding the Actor and Critic classes (Figure 3.3):

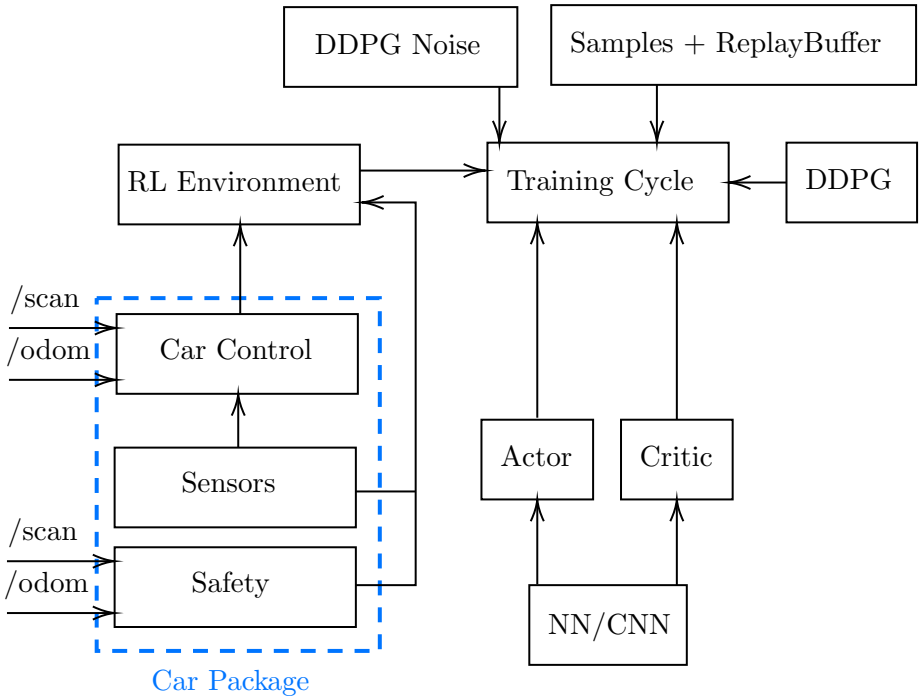

FIGURE 3.3: DDPG controller structure

The agent inputs and outputs for both the DQN and DDPG controllers are as introduced in Figure 3.4. The data from the `\odom` and `\scan` topics is obtained through the `get_car_state()` function, then is processed and stored in a state. The state stored in the replay memory is then fed to the `behavior_predict(state)` function, which returns

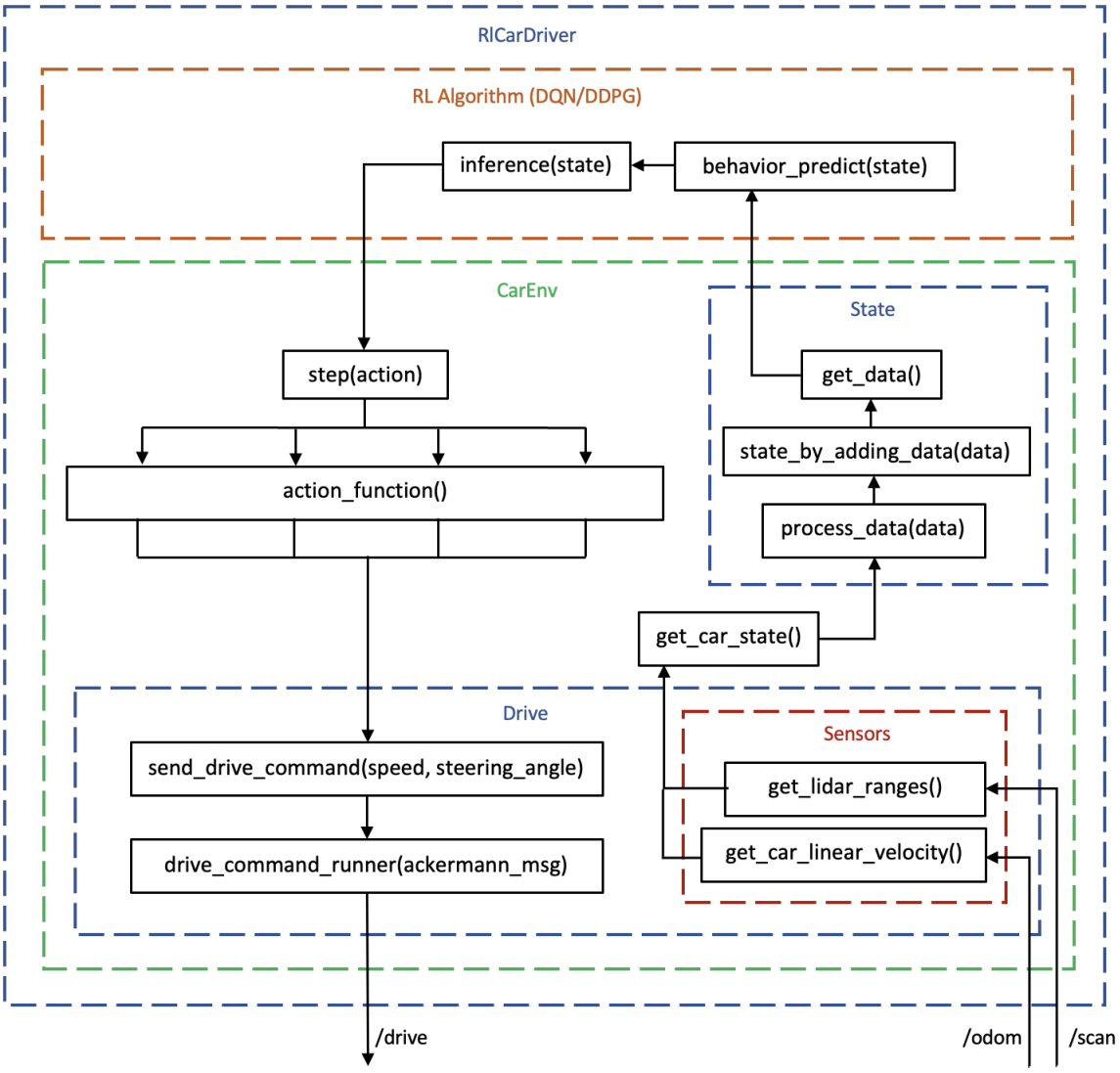

FIGURE 3.4: Functions involved in running an epoch and their inputs/outputs

an array of 4 float values between 0 and 1, corresponding to each possible action. The action with the highest score is then sent to the `step(action)` function, which calls the corresponding action function, for example `forward()`. This action function then sends a desired speed and steering angle to the `send_drive_command` function, which creates an Ackermann message. This message is finally published to the `drive` topic by the `drive_command_runner(ack_msg)` function.

Requirements E3, E5, E7 and E8 will be fulfilled using log files, listening to the `/odom` topic to calculate the different metrics. For requirement E8 (Sim2Real gap assessment), the data will come from the Inertial Measurement Unit (IMU).

Several Python libraries will be used for the DQN and DDPG controllers:

- TensorFlow 2.6.0: open source library with flexible architecture enabling high-performance computation for machine learning and deep learning applications
- Keras 2.7: a scalable framework built on TensorFlow offering simple deep learning implementations; it will be used to implement CNNs.
- Numpy: package providing high-performance computation capabilities for multi-dimensional arrays.
- Matplotlib 3.5.3: library offering visualisation tools.

A few non-functional software requirements will also be implemented (Table 3.3):

Reference	Name	Description	Priority
N1	Documentation	The different parameters used should be listed for reproducibility; instructions for running the code should be provided in a README file.	High
N2	Compatibility	Instructions will be provided on how to run the controllers on other autonomous racing car platforms.	Low
N3	Licensing	The use of the MIT License should be documented in the README file (Section 4.2).	Medium

TABLE 3.3: Non-fuctional software requirements

3.5 Preliminary Investigations

To avoid encountering too many technical issues with the F1Tenth simulator and ROS, I have made several preliminary investigations:

- I have installed and tested the F1Tenth default simulator on my Ubuntu VM alongside ROS and Catkin.
- I have also found a way of installing the TensorFlow library on an ARM64 architecture by building the library from source.
- I have also looked at how different maps can be imported in the simulator using the YAML format and editing the `simulator_launch` file.
- I have looked at how I can make the ROS simulator run faster than real-time, which will be helpful when training the DQN and DDPG controllers; this is achievable using the `/clock` topic with the `/use_sim_time` parameter set to `true`.
- I have looked at the ROS `pid` module, that I could use to tune the parameters of the wall following controller.

These investigations have not directly answered any experimental or software requirements, but they will save time for the project's next steps.

3.6 Evaluation

The goal was first to assess two different parameters: the speed and smoothness of the controller using the metrics introduced in Table 3.4; They are inspired by the metrics presented in Hermansdorfer et al. [2020]. They can all be measured directly from ROS topics, both in the F1Tenth simulator and on the actual car. The best-performing algorithm will be the one that maximises the average speed and minimises the other metrics.

Metric	Description
T_{avg}	Average time to complete a full lap
V_{avg}	Average speed over a full lap
V_{max}	Maximum speed reached
A_{avg}	Average acceleration
A_{max}	Maximum acceleration
D_{max}	Maximum deceleration

TABLE 3.4: First set of metrics for performance evaluation

After some consideration, it was decided to update the metrics to make them more relevant to the evaluation of the controllers (Table 3.5). Firstly, the average speed was removed as it is redundant with the time required to complete a lap, and the maximum reached speed was removed as it is always equal to the car's max speed. Secondly, the average positive and negative accelerations were introduced to replace the average acceleration, which is almost equal to 0. Finally, the average minimum LiDAR range was introduced to better assess the controller's safety, with higher values meaning that the controller is, on average, further away from the closest obstacle.

Metric	Description
T_{avg}	Average time to complete a full lap
A_{avg}	Average acceleration
D_{avg}	Average deceleration
A_{max}	Maximum acceleration
D_{max}	Maximum deceleration
$LiDAR_{min}$	Average minimum LiDAR distance

TABLE 3.5: Updated set of metrics for performance evaluation

I will consider this project to have successfully answered the research questions introduced in Section 1.1 if:

- The controllers implemented have been compared using the metrics from Section 1.2, their ability to generalise to new race tracks and the training time required.
- The controllers have been compared when looking at the Sim2Real gap using the metrics from Section 1.2.

I will consider the software part of the project successful if all the high and medium-priority functional and at least some of the non-functional software requirements are implemented.

Chapter 4

Professional, Legal, Ethical, and Social Issues

4.1 Professional Issues

This project will respect the British Computer Society (BSC) Code of Conduct.
The Python code associated with this project will be published on a public GitHub repository, as well as the LaTeXcode of this research report. The code will be precisely commented, and the parameters used to obtain results will be listed in detail. The goal is to have reproducible results and make it easy for other people to integrate this code into their own applications.
The progress of the project and the obstacles faced will be documented in the repository, with regular commits and maximum transparency. We will strive to follow best practices during the development of all the components of the codebase, trying to make the code as explicit and objective as possible. The PyStyleGuide (introduced in [PyStyleGuide]) will be followed when writing Python ROS code. Python code will follow the PEP 8 conventions ([PEP8]), valuing consistency and readability. A short summary of the PEP 8 recommendations is as follows:

- `package_name`
- `ClassName`
- `field_name`

- `method_name`
- `_private_something`
- 4 spaces for indentation

Code from other sources, like open-source packages and libraries, will be credited and listed in the repository, independently of the license.
The final objective is to develop code easily reusable in other contexts, for example, to be embedded in systems other than the F1Tenth platform.

4.2 Legal Issues

There is no intent to monetise this project; this is why it was decided to use the MIT License for the codebase. This license is very permissive, not only for open-source use but also for profit (businesses). Anyone will be able to use the code to develop third-party software. MIT License was chosen for two reasons:

- The license is defined in a very straightforward way: it is easy to understand the terms and conditions
- The license makes it easier to link code with free open source software and proprietary closed source software and is compatible with other licenses such as GPL

No dataset will be used in this project as we will generate our own data. The maps used for training and evaluating the controllers were taken from Brunnbauer et al. [2022], found at the following link: https://github.com/CPS-TUWien/racing_dreamer/tree/main/docs/maps.

4.3 Ethical Issues

No ethical issues are directly associated with this project; however, autonomous racing is part of the more general field of autonomous driving, which features several realistic problems presented in Hansson et al. [2021]. Much of those issues are related to the concept of responsibility: when all occupants of the vehicle become passengers, who is

responsible for their safety and the safety of other road users? Other ethical problems include over-reliance on automatic safety systems, hacks, and data privacy.

4.3.1 Social Issues

The ORBIT AREA 4P Framework ([ORBIT]) is a tool to help researchers in Information and Communications Technologies (ICT) to work responsibly. The framework asks several questions about the research, divided into four categories (anticipating, reflecting, engaging and acting) and along four axes (the process, the product, the purpose and the people). Two questions are particularly relevant to this project:

- **Is the planned research methodology acceptable?** The project does not cause any lab health & safety issues to people. To ensure the safety of the F1Tenth car, a module will be implemented to the car package (see Subsection 3.3.1) to prevent collisions when possible. The supervisor accepted an ethical approval form, and the methodology was introduced in 3.
- **What are the viewpoints of a wider group of stakeholders?** Being part of the LAIV at Heriot-Watt University, I was able to get feedback from a wider group of researchers and academics through a presentation on RL and weekly meetings with my supervisor, Luca Arnaboldi and other academics.

Chapter 5

Project Management

5.1 Risk Management

The list of expected risks and proposed mitigations is given in Table 5.1. As explained in the table, there are three main risks: having longer than expected training times, being incapable of replicating experimental results and having compatibility issues on my Ubuntu ARM64 VM. The primary approach to minimising those risks will be to prioritise the most complicated aspects of the project; some of that work was done in the preliminary investigations, as explained in Section 3.5. This will let me identify problems early in the process, and I will be able to ask for help and advice before I run out of time.

5.2 Project Plan

The GANT chart introduces the project plan (Figure 5.1). Because I don't have any constraints related to equipment availability or human evaluations, I was able to plan everything around my own constraints, especially the time needed to train the controllers for DQN and DDPG.

Risk	Likelihood	Impact	Mitigation
Unable to replicate results of Bosello et al. [2022] and Ivanov et al. [2020]	Medium	High	Allocate highest priority to task in the project plan; ask supervisor for advice
Involuntary plagiarism by missing something in the wider literature	Low	High	Do regular literature reviews on specific points; ask supervisor for input
Training the RL controllers takes longer than expected	High	Medium	Optimise the code, reduce the size of NNs, use other hardware, possibly Cloud computing or Google Colab
Loss of Python and/or LaTeX files due to hardware failure or human error	Low	High	Save files regularly on the Cloud or an external drive
Compatibility errors with ML Python libraries on my Ubuntu ARM64 VM setup	Medium	High	Ask for help on StackOverflow, use different machine or libraries if necessary
Tuning manually the PID parameters K_p, K_i and K_d for the Wall Following controller doesn't work	Medium	Medium	Use MATLAB or possibly the ROS `pid` package
Applying the controllers on the physical car doesn't work: it damages the car, there are delays in communications with the actuators and sensors, the LiDAR data is not as perfect as in the simulator	Medium	High	Increase speed progressively, implement a safety module in the ROS code to stop the car if it gets too close to a wall, make sure the walls of the track are not made out of reflective materials

TABLE 5.1: Risk assessment approach

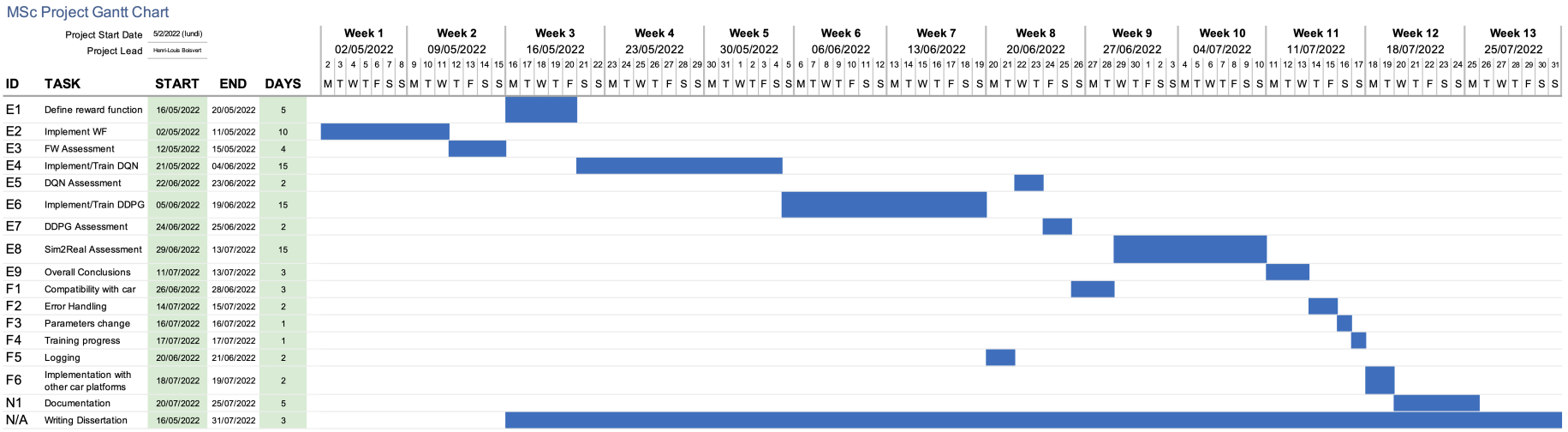

FIGURE 5.1: Gantt Chart of the project

Chapter 6

Experimental Protocol

The experimental protocol was designed around the nine experimental requirements presented in Subsection 3.3.1 and corresponding to the four research hypotheses introduced in Section 3.2.

When starting the project, I found out that the planning I had introduced in Section 5.2 could not work: the reward function requirement (E1) cannot be worked on before the DQN implementation (E4) and the DDPG implementation (E6), because the reward function may depend on the controller implementation and outputs. Furthermore, because the DQN controller outputs discrete actions and the DDPG controller outputs continuous actions, they would require different reward functions. The new plan was thus to start with the second experimental requirement E2 (implementing the wall following controller), then assess it (E3) and then implement and train the Deep Q-Network controller (E4), defining the reward function at the same time (E1). I would then do the same with the DDPG controller and finish with the Sim2Real gap assessment (E8) and the overall conclusions (E9).

As explained in Section 3.2, the DQN and DDPG controllers would be trained on the "RedBull Ring" racetrack from Bosello et al. [2022] for requirements E4 and E6. The assessment of all three controllers in the simulator (wall follower, DQN and DDPG) would then be done on the RedBull track as well as the Silverstone (Appendix A.2) and Monaco (Appendix A.3) racetracks. Those tracks are scaled to one-tenth of their actual size to be at the same scale as the car.

Those specific tracks were chosen for several reasons. Firstly I thought the Red Bull Ring track to be a good training map because it presents most of the challenges that the

controller will encounter on other racetracks: a sharp turn (n°3 Remus, see Appendix A.1), wide turns (6,7) and two long straight lines, all of that on a very short circuit, which I think makes for good training data. The second track, Silverstone, is much longer with very long curves and some chicanes, which represents a different challenge: many small adjustments in the long curves and swift adjustments for the chicanes. The Monaco track also offers a different challenge, with many hairpin turns (5,6 and 16, see Appendix A.3).

6.1 Modifications to the experimental requirements

After trying to implement the DDPG controller, I decided to drop experimental requirements E6 (Implement and train DDPG) and E7 (DDPG assessment). I had previously reviewed a DDPG implementation suitable for autonomous racing (Ivanov et al. [2020]), but I found it too time-consuming to make it work alongside the F1Tenth simulator. Furthermore, the number of hyper-parameters of the DQN controller made it very time-consuming to complete experimental requirement E4 (6 different DQN controllers had to be trained for a total of 24 DQN assessments on all racetracks). I had assigned a lower priority to the DDPG requirements, and dropping them was considered not to be too detrimental to the overall dissertation because the results from the Wall Following and DQN controllers were sufficient to answer research hypotheses 1, 2, 3 and 4.
After some consideration, I decided to also assess the controllers (experimental requirements E3 and E5) on a fourth track (see Appendix A.4). This track has a very simple oval shape, which makes it easy to reproduce in real life using flexible tubing; this would make the Sim2Real gap assessment (experimental requirement E8) much more meaningful because the metrics introduced in Section 3.6 would be comparable on a whole lap of the track and not just on a small portion of it. I defined the shape of the track to make it as compact as possible but with a sufficient track width.

6.2 GitHub repository

The GitHub repository is structured with folders for each controller, a folder for the F1Tenth simulator, one for the experimental results as Excel files, and one for the Latex code of this dissertation. Several Readme files have been written to explain the procedures and parameters for each experiment.

The `EXPERIMENT_PARAMETERS.md` file explains in detail the process I followed to set up the controllers with the F1Tenth simulator. I found it very important to use the 2.0.0 version of the Tensorflow library; this was quite challenging on my laptop as my CPU uses the ARM architecture, and there is no easy way to install the TensorFlow library on an ARM processor. After asking on StackOverflow, I was advised to use the pre-built binary available at https://github.com/PINTO0309/Tensorflow-bin#usage, which is built for the Raspbian Linux distro but also works on Ubuntu 20.04.3.

6.3 Wall Following implementation

The second experimental requirement (E2) was to implement and tune the wall following controller in the simulator. My implementation is based on the code shown in the official F1Tenth labs and uses the equations introduced in Subsection 2.16.

The tuning of the PID controller was done manually as follows: first, I set the three gains k_p, k_d and k_i to zero. Then, I increased k_p until the response to a disturbance was a steady oscillation; I then increased k_d until no oscillations were left. I then repeated steps 2 and 3 until k_d didn't stop the response from oscillating. I finally increased k_i until the response reached the set point within only two oscillations. This last step is a compromise between speed and accuracy: if k_i is too small, the controller will be very fast but will overshoot the angle, and if it is too big, the car will react much slower. More information about the process and the parameters can be found on the project's GitHub at https://github.com/HL-Boisvert/MSc_Project_HW.

Once E2 was completed, I had to gather the experimental values for E3. I decided to export the data to a .csv file, which can then be processed in Excel. This was done using the Python `csv` package, which offers much flexibility for exporting the ROS data to .csv format. The export script is implemented as a Python class, instantiated by the WallFollow class.

6.4 Deep Q-Network implementation

The Deep Q-Network controller used to obtain experimental results is extensively based on the implementation from Bosello et al. [2022]. I will first explain the code's architecture and then introduce the differences I've made to the original code.
The UML diagram of the final code is introduced in Figure 6.1.

The code is structured around three main classes: `RLCarDriver`, `CarEnv` and `DeepQNetwork`.
The `RLCarDriver` class is responsible for three things: setting up the output and logging directory, parsing the parameters from the user input and running the `run_epoch()` function inside of a `while(not stop)` loop. The `run_epoch()` function does several things. First, it gets the state from the environment, then updates the value of epsilon; it then computes the action of the agent according to the epsilon-greedy policy, calling the `DeepQNetwork` class if necessary; finally, it makes a move, and records the experience in the replay memory.
The `CarEnv` class is the interface which enables communication between the F1Tenth simulator and the controller; it instantiates the `State`, `Drive`, `Sensors` and `SafetyControl` classes. The `Drive` class publishes the Ackermann messages to the `\drive` topic, according to the action sent by the CarEnv class (see Figure 3.4). The Sensors class subscribes to the `\odom` and `\scan` topics for them to be stored in a state: the `State` class stores the LiDAR distance values as well as the speed of the agent.
The `SafetyControl` class has not been modified from Bosello et al. [2022] and sends the `stop` action to the `Drive` class if the car gets too close to an obstacle.
The `DeepQNetwork` class implements the actual DQN algorithm. It creates the behaviour and target NNs and defines a training step according to the pseudo-code introduced in Subsection 2. It takes as an input the LiDAR and velocity data from a specific state stored in the Replay Memory as a sample and outputs an integer representing an action (like going forward or turning left) through the `inference(state)` function.
The subscribers, publishers and topics are shown in Figure 6.2.

I have made several changes to the implementation from Bosello et al. [2022]: I've added a class for exporting data to CSV format to then use it in Excel; I've made it possible to choose between different reward functions; I've made the saving and loading of a

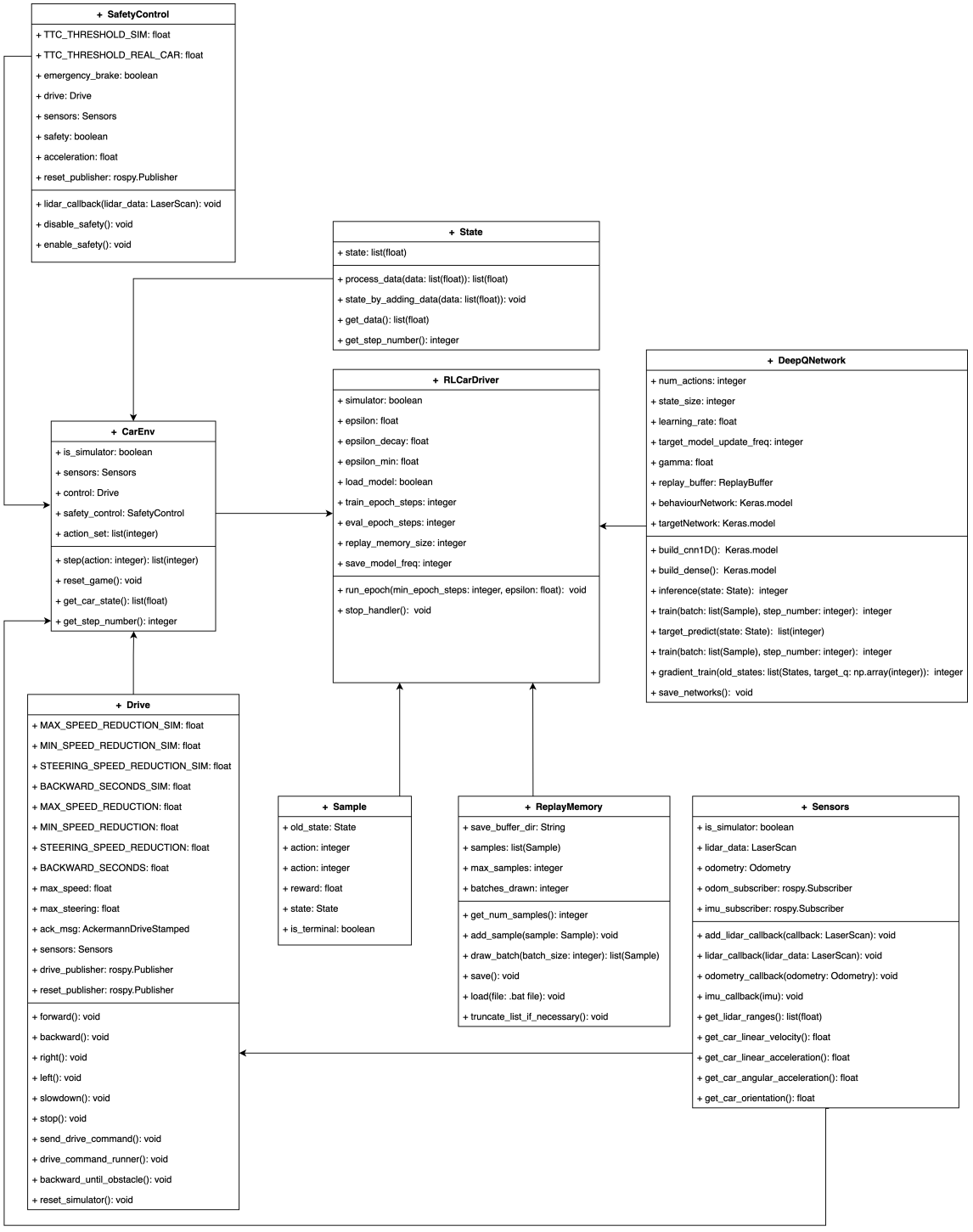

FIGURE 6.1: UML diagram of the Deep Q-Network implementation based on Bosello et al. [2022])

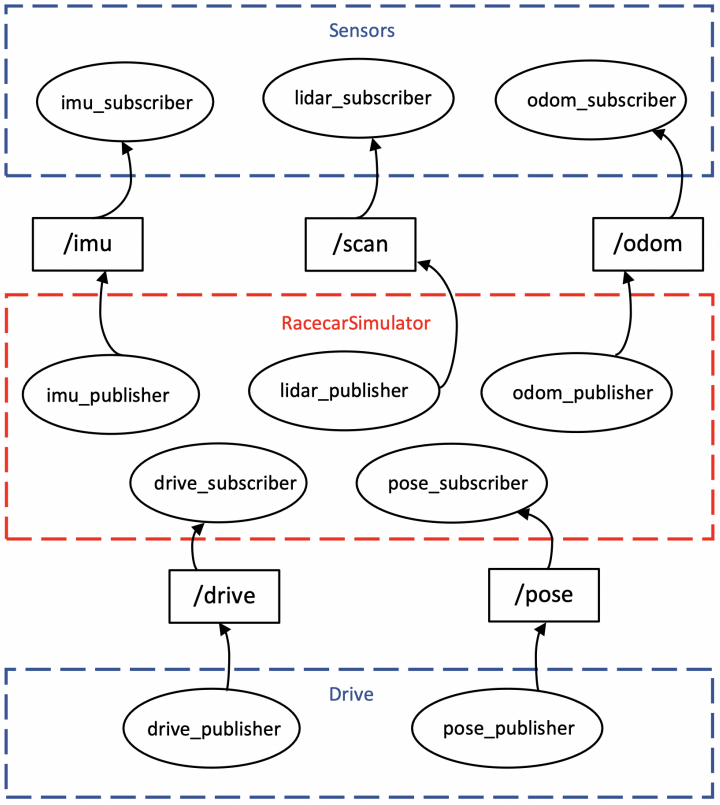

FIGURE 6.2: Subscribers, publishers and topics for the Deep Q-Network implementation

trained network more straightforward, and I've removed several parameters and classes that were not needed for the experiments.

6.5 Reward functions

The first requirement (E1) was to define a reward function that incentivises a safe, smooth and fast control of the car. Several reward functions were considered for the DQN controller.

Before defining the reward function, I had to define the actions that the RL agent would be able to perform. The controller can send two types of commands to the agent through an AckermannDrive Message: a desired forward speed command (m/s) and a desired steering angle command (radians). Thus, the reward function must incorporate at least one of those two variables.

The problem is that DQN can only output discrete actions, whereas the steering and speed outputs of the car are continuous. A simple solution is to discretise the action space, for example, by replacing all possible steering angles with just a few increments

and doing the same for speed. The car can steer from around -40° to +40°: with 4° increments, this gives us 20 steering outputs; the car can go from 0 to 5 m/s, with 1 m/s increment, this gives 5 possible speed outputs: this results in 100 possible outputs. The controller is now facing the "curse of dimensionality" problem. There are much more states to explore, and the training time will increase dramatically.
To avoid this issue, I decided to instead limit the controller outputs to just 4: going forward at full speed, turning left or right at reduced speed and maximum steering angle, and keeping the same steering angle at reduced speed.
This way, the controller is less smooth but can learn faster; if necessary, adding a few other outputs (turning slightly left/right) with different steering angles and speeds is easy.

6.5.1 Reward Function n°1

The first attempt was to have a reward purely proportional to the speed of the car, assuming naively that the controller could achieve a safe and smooth behaviour just from the speed data: for step t, $r(t) = v(t) \cdot \frac{1}{V_{max}}$. This resulted in a few oscillations and frequent collisions because the agent is not looking at the LiDAR data and thus is not incentivised to slow down enough when turning. It's possible that with enough training steps, the agent might still achieve a safe behaviour with this reward function: collisions still result in a lower reward because the agent stops when hitting an obstacle; however, with our time limits, the behaviour was unsatisfactory.
I tried something similar and assigned a specific reward for each possible action: greatest for going forward, less for going left or right, and no reward for slowing down (Equation 6.1):

$$\begin{cases} r(t) = 1 & \text{if } \texttt{going\_forward} \\ r(t) = 0.5 & \text{if } \texttt{going\_left} \text{ or } \texttt{going\_right} \\ r(t) = 0 & \text{if } \texttt{slowing\_down} \end{cases} \tag{6.1}$$

This reward function resulted in similar behaviour, with a few oscillations and unsafe behaviour. This was to be expected because the agent is still not being rewarded for

keeping a good distance from the track walls.

6.5.2 Reward functions n°2 and 3

To make the behaviour safer, I then took into account the LiDAR data (Equation 6.2):

$$r(t) = v(t) \cdot \frac{1}{V_{max}} \cdot W_1 + min(lidar_{values}) \cdot W_2 \tag{6.2}$$

with W_1 and W_2 weights for respectively the speed and LiDAR rewards. I tried with $W_1 = 0.5, W_2 = 1$, which is Reward Function n°2, and $W_1 = W_2 = 1$ which is Reward Function n°3 (see Section 7.1). The function n°2 makes the safety requirement more important than the speed requirement, whereas the function n°3 gives them equal importance. Some fine-tuning of the weight values could be done, depending, for example, on the reliability of the LiDAR data, but this is outside this project's scope.

6.6 Speeding up the simulation

Because experimental requirement E4 necessitates training 6 different DQN controllers, it was decided to limit the training time of a controller to around 24h. After assessing that the controllers could not achieve an entire loop of the Monaco and Silverstone tracks within that training time, I chose to use a Clock Server to speed up the training.
Usually, the ROS nodes run using the computer's system clock as the time source; however, it is also possible to have ROS instead listen to a simulated clock so that the agent can perceive time as going faster: this is what I decided to do. The process is more straightforward than expected; first, I had to tell ROS to use the simulated clock time; this is done by setting the parameter `use_sim_time` in the `simulator.launch` file to "True". Then, Clock Server was run, publishing to the `\clock` topic at a particular frequency. It was found that if the frequency is not high enough compared to the speed multiplier, the controller will "skip" some steps when publishing, and the car will become uncontrollable and jump at random. After some testing, it was established that publishing to the `\clock` topic at 100Hz with a 3x speed multiplier works fine, and increasing the speed multiplier above 3 seemed useless as the hardware started to struggle. The Clock Server was implemented as a single .py file with only one function

for publishing; it needs to be started before the F1Tenth simulator. More information on Clock Servers can be found at http://wiki.ros.org/Clock.

6.7 Assessment protocol

Metrics are measured as an average over 10 laps; to determine if a controller performs better than another one, the metrics are used to compare 3 criteria that could be of interest to the user:

1. The speed (Time to complete a lap, the lower the better)
2. The smoothness (Average and maximum acceleration/deceleration, the closest to zero the better)
3. The safety (Average minimum distance to a wall, the higher the better)

The metrics are then compared between controllers in two ways to account for statistical errors:

1. Using a 2-standard-deviation range (95% confidence interval) calculated from the variances of each ten-sample batch: this interval is represented by the error bars in the figures in Chapter 8.
2. Using the P-Value method to reject or support the Null Hypothesis.

To verify Hypothesis n°1 (*Deep Reinforcement Learning provides a real advantage, quantified by reliable metrics, for controlling an F1Tenth car over human control and PID-based methods like wall following.*) and Hypothesis n°2 (*CNNs offer better results over selected metrics than NNs for autonomous racing on the F1Tenth platform using data from a UST-10LX LiDAR.*), the metrics are compared for all controllers on the RedBull track using 10-sample batches. For those two hypotheses to be verified with certainty, there must not be any overlapping of the 95% confidence intervals for each sample.
To verify Hypothesis n°3 (*DRL-based controllers can generalise to new race tracks without affecting too much the metrics.*), the metrics of the DQN controllers trained on the RedBull track and tested on the Monaco track are compared to those of the controllers

trained and tested on the Monaco track. For the hypothesis to be verified, the values must be in each other 2-standard-deviations range.

To verify Hypothesis n°4 (*The sim2Real gap is negligible (minor changes of the metrics) and doesn't affect the performance of the controllers implemented.*), the metrics were measured on the actual car running on the oval track from Appendix A.4 (average over 10 laps) and compared to those obtained in the simulator. For the hypothesis to be verified, the values must be in each other 2-standard-deviation range.

Chapter 7

Experimental Results

All the experimental data is available in an Excel file at https://github.com/HL-Boisvert/MSc_Project_HW. The data was extracted from ROS using the data scripts in the `Wall Following` and `DQN` folders and has been processed to be easily understandable. Spreadsheets for answering the hypotheses were also included in the Excel file.

7.1 Experimental Requirements E3 and E5

Experimental Requirement E3 consisted of assessing the performance of the Wall Following controller according to the updated metrics on the four simulated tracks.
Regarding Experimental Requirement E5 (DQN assessment), as explained in the GitHub repository, all 6 controllers were trained for 1 million steps (around 20h), using the same hyper-parameters. Values replaced with "/" indicate that the lap couldn't be completed by the car.
It was decided to compare the performance of all controllers one track at a time: Tables 7.1, 7.2, 7.3 and 7.4 introduce the results for each track; the detailed data is available in the Excel file.

To answer Hypothesis n°3, the CNNs trained on the Monaco Track were then compared to those trained on the RedBull track when racing on the Monaco Track; the results for all metrics are introduced in Table 7.5.

Controller	Total Time (s)	Average Positive Acceleration (m/s^2)	Average Negative Acceleration (m/s^2)	Maximum Acceleration (m/s^2)	Maximum Deceleration (m/s^2)	Average Minimum LiDAR Range (m)
Wall Following	26.86	1.17	-3.16	4.87	-7.90	0.84
NN R1	36.24	1.00	-3.61	5.39	-6.51	0.84
NN R2	27.81	1.31	-3.35	5.09	-5.81	0.82
NN R3	29.15	0.82	-4.54	3.49	-6.71	0.81
CNN R1	35.24	0.82	-3.48	6.35	-5.57	0.81
CNN R2	25.17	0.72	-3.23	4.97	-6.72	0.80
CNN R3	24.20	1.09	-2.23	5.38	-6.53	0.77

TABLE 7.1: Experimental results on the RedBull track

Controller	Total Time (s)	Average Positive Acceleration (m/s^2)	Average Negative Acceleration (m/s^2)	Maximum Acceleration (m/s^2)	Maximum Deceleration (m/s^2)	Average Minimum LiDAR Range (m)
Wall Following	50.68	1.10	-2.41	3.59	-7.90	0.84
NN R1	/	/	/	/	/	/
NN R2	53.62	0.87	-3.73	5.61	-6.69	0.77
NN R3	56.34	0.81	-3.55	4.71	-2.90	0.80
CNN R1	/	/	/	/	/	/
CNN R2	46.74	0.90	-1.29	5.24	-6.58	0.78
CNN R3	45.01	0.88	-1.09	4.31	-2.91	0.78

TABLE 7.2: Experimental results on the Monaco track

Controller	Total Time (s)	Average Positive Acceleration (m/s^2)	Average Negative Acceleration (m/s^2)	Maximum Acceleration (m/s^2)	Maximum Deceleration (m/s^2)	Average Minimum LiDAR Range (m)
Wall Following	51.92	0.89	-3.11	4.06	-7.52	0.65
NN R1	/	/	/	/	/	/
NN R2	61.22	2.43	-1.21	4.61	-5.22	0.81
NN R3	63.41	1.22	-1.57	5.24	-6.79	0.80
CNN R1	/	/	/	/	/	/
CNN R2	49.78	1.01	-2.13	5.79	-5.48	0.81
CNN R3	50.44	0.86	-1.43	4.59	-4.83	0.78

TABLE 7.3: Experimental results on the Silverstone track

Controller	Total Time (s)	Average Positive Acceleration (m/s^2)	Average Negative Acceleration (m/s^2)	Maximum Acceleration (m/s^2)	Maximum Deceleration (m/s^2)	Average Minimum LiDAR Range (m)
Wall Following	3.80	1.23	-4.04	4.90	-7.51	0.62
NN R1	3.26	1.31	-0.87	3.51	-1.73	0.76
NN R2	3.19	1.22	-0.96	3.40	-1.93	0.81
NN R3	2.98	1.21	-0.96	4.51	-1.62	0.78
CNN R1	3.01	1.36	-0.51	3.82	-1.89	0.78
CNN R2	3.42	1.21	-1.82	2.98	-1.25	0.79
CNN R3	2.95	1.26	-0.74	3.91	-1.42	0.78

TABLE 7.4: Experimental results on the oval track

Controller	Total Time (s)	Average Positive Acceleration (m/s^2)	Average Negative Acceleration (m/s^2)	Maximum Acceleration (m/s^2)	Maximum Deceleration (m/s^2)	Average Minimum LiDAR Range (m)
CNN R2 Monaco Trained	42.28	0.93	-1.25	5.18	-6.67	0.80
CNN R3 Monaco Trained	40.91	0.84	-1.14	4.34	-2.69	0.83
CNN R2 RedBull Trained	46.74	0.90	-1.29	5.24	-6.58	0.78
CNN R3 RedBull Trained	44.98	0.89	-1.10	4.30	-2.94	0.77

TABLE 7.5: Experimental results on the Monaco Track comparing DQN controllers trained on Monaco or RedBull tracks

7.2 Experimental Requirement E8

Experimental Requirement E8 consisted of running the Wall Following and DQN controllers on the physical car, racing on the oval race track built using flexible tubes, with the goal of evaluating the Sim2Real gap. It was found that the height of a single flexible tube was not enough to be reliably detected by the LiDAR. Thus, a second perimeter of tubes had to be taped on top of the first one (Figure 7.1).

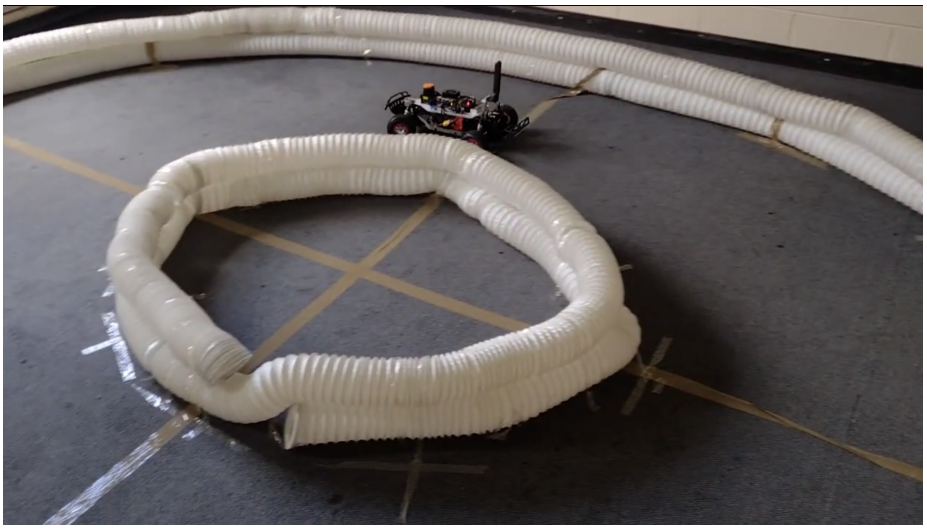

FIGURE 7.1: Track with the car in starting position

7.2.1 Wall Following controller

It quickly appeared that the tuning of the gains k_p, k_d, and k_i in the simulator was not adapted to the physical car, which showed a clear Sim2Real gap; the PID was tuned again in the same way as explained in Chapter 6, and detailed results are available in the Excel file. It also appeared the controller was not performing properly at a speed of 5 m/s: the max speed was changed to 4 m/s, and the simulator experiments were performed again with the new max speed.

7.2.2 DQN

The same problem was encountered on the car when installing TensorFlow on the Jetson and was solved by following the instructions at https://docs.nvidia.com/deeplearning/frameworks/install-tf-jetson-platform/index.html.

After some testing, it was found the odometry wasn't reliable and couldn't be used as input for the NN and CNN controllers, meaning new controllers had to be trained. Because of time constraints, only the most performant controller (CNN with reward function n°3) was retrained. Because the odometry couldn't be relied upon, the metrics that depended on the speed measurement couldn't be assessed, which only left the lap time and average minimal LiDAR distance metrics; this was considered enough to answer Hypothesis n°4.

Finally, there was an issue with the controller in the last turn of the lap which made

the car collide with the tubes; the issue happened each time a lap was attempted, even when adding a higher obstacle and changing the lighting in the room. In the end, it was decided to use the data only on 3 quarters of a lap and then extrapolate. The problem couldn't really be explained, it may be linked to the training data, the LiDAR input, the hardware itself or several of those reasons at the same time.
The detailed data is available in the Excel file, and the main results are introduced in Table 7.6 (averages over 3 laps).

Controller	Total Time (s)	Average Minimum LiDAR Range (m)
Manual Driving	5.47	0.72
Wall Following	3.75	0.76
CNN RW3	3.55	0.77

TABLE 7.6: Experimental results on the physical oval track

Chapter 8

Conclusions

In this dissertation, we provide two main contributions to the field of autonomous racing:

1. We introduce a simple framework for comparing autonomous racing controllers using three key metrics: speed, safety and smoothness and accounting for statistical error using the P-Value method.
2. With those metrics, we look at four key problems:
    - How DQN performs compared to Wall Following (H1)
    - How DQN using CNNs perform compared to NNs (H2)
    - The ability of DQN to generalise to unseen tracks (H3)
    - The ability of DQN to overcome the Sim2Real gap (H4)

We will first discuss those four hypotheses in Section 8.1, then make some general conclusions in Section 8.2, and finally propose possible future improvements to this project in Section 8.3.

8.1 Reflexions

8.1.1 Hypothesis n°1

When looking at the comparison between the lap time, average minimal LiDAR distance and average acceleration/deceleration metrics on the RedBull track (Figures 8.1, 8.2,

8.3 and 8.4), we can draw 3 conclusions regarding Hypothesis n°1 (*Deep Reinforcement Learning provides a real advantage, quantified by reliable metrics, for controlling an F1Tenth car over human control and PID-based methods like wall following.*):

- Regarding the speed metric (time to complete a full lap), Hypothesis n°1 appears to be verified for 2 DQN controllers: the CNN with reward functions n°2 and 3, with the latter performing the best. The hypothesis may also be verified for a NN with reward function n°2, but the experimental uncertainty is too high to confirm this result.
- Regarding the safety metric (average minimum LiDAR distance), Hypothesis n°1 appears to be verified for all DQN controllers apart from the NN with reward function n°1. The best performing controllers are the CNNs, with the reward function n°3 being the best.
- Regarding the smoothness metric (average acceleration/deceleration), Hypothesis n°1 is only verified by the CNN with reward function n°3 when looking at the average deceleration metric but is verified by all DQN controllers apart from CNN with reward function n°2 when looking at the average acceleration.

Those conclusions were made by looking at the 95% confidence intervals represented by the errors bars (in Figures 8.3, 8.4, 8.2, 8.1, 8.8, 8.9, 8.7, 8.6) as explained in Chapter 6. Hypothesis n°1 can then be checked using the P-Value method (more details are available in the Excel file at `https://github.com/HL-Boisvert/MSc_Project_HW`):

- **Null Hypothesis (H0)**: CNNs provide no real advantage with reward functions 2 and 3 over Wall Following.
- **Alternative hypothesis (H1)**: CNNs provide a real advantage with reward functions 2 and 3 over Wall Following.

Using the standard value of Alpha = 0.05 and an independent two-tailed T-test, we obtained the following results:

- Time metric: comparing the 10-lap sample with reward function n°2 to the 10-lap sample with Wall Following gives $p - value = 1.81 \cdot 10^{-12} < 0.05$, which means the result is significant, and the Null Hypothesis can be rejected. With the reward function n°3, we get $p - value = 7.29 \cdot 10^{-16} < 0.05$: the result is also significant.

- verage acceleration metric: comparing the 10-lap sample with reward function n°2 to the 10-lap sample with Wall Following gives $p - value = 1.14 \cdot 10^{-22} < 0.05$, which means the result is significant, and the Null Hypothesis can be rejected. With the reward function n°3, we get $p - value = 6.56 \cdot 10^{-10} < 0.05$: the result is also significant.

- Average deceleration metric: comparing the 10-lap sample with reward function n°2 to the 10-lap sample with Wall Following gives $p - value = 0.0094 < 0.05$, which means the result is significant, and the Null Hypothesis can be rejected. With the reward function n°3, we get $p - value = 6.64 \cdot 10^{-23} < 0.05$: the result is also significant.

- Average minimal LiDAR distance metric: comparing the 10-lap sample with reward function n°2 to the 10-lap sample with Wall Following gives $p - value = 2.83 \cdot 10^{-18} < 0.05$, which means the result is significant and the Null Hypothesis can be rejected. With the reward function n°3, we get $p - value = 1.42 \cdot 10^{-22} < 0.05$: the result is also significant.

To conclude, the only DQN controller which verifies Hypothesis n°1 for the speed, safety, and smoothness metrics is the CNN with reward function n°3. This shows that a DQN controller has to be trained using a carefully chosen reward function to be efficient and that with the limited training time used in the experimental protocol, NNs don't verify the hypothesis.

However, it is likely that for a longer training time, the DQN would have been able to perform better, as the average reward during an evaluation epoch of several of the controllers was still increasing towards the end of the training, especially CNNs with reward functions 2 and 3 (Figure 8.5). For those two controllers, the reward increased by over 30% during the last 30 evaluation epochs, unlike the other controllers, which didn't show any significant reward increase over that time.

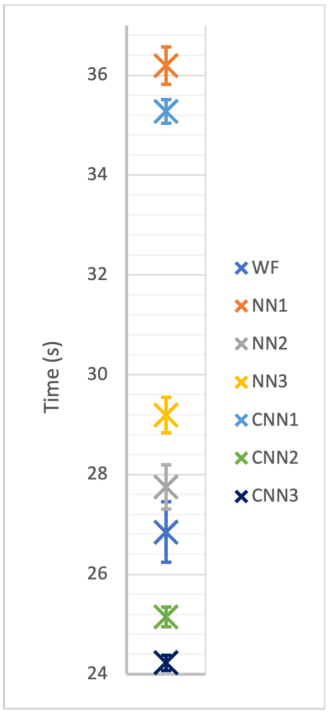

FIGURE 8.1: Lap time comparison for controllers on the RedBull track (from Excel file)

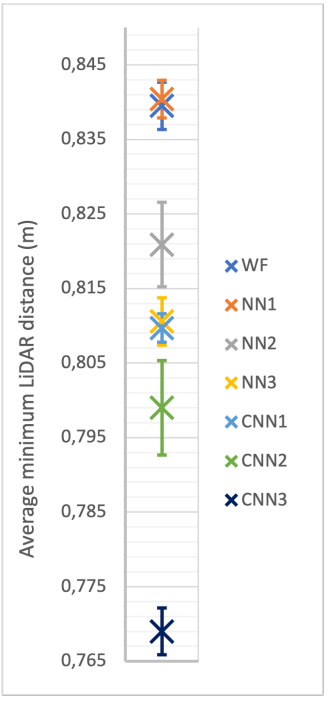

FIGURE 8.2: Average minimal LiDAR distance for controllers on the RedBull track (from Excel file)

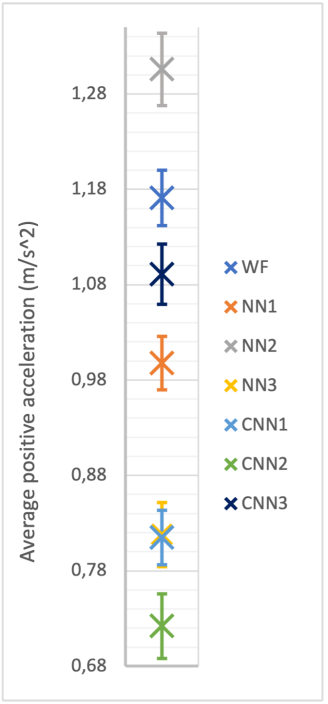

FIGURE 8.3: Average acceleration comparison for controllers on the RedBull track (from Excel file)

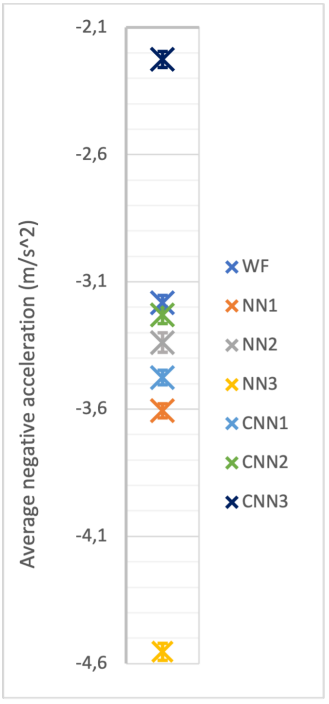

FIGURE 8.4: Average deceleration comparison for controllers on the RedBull track (from Excel file)

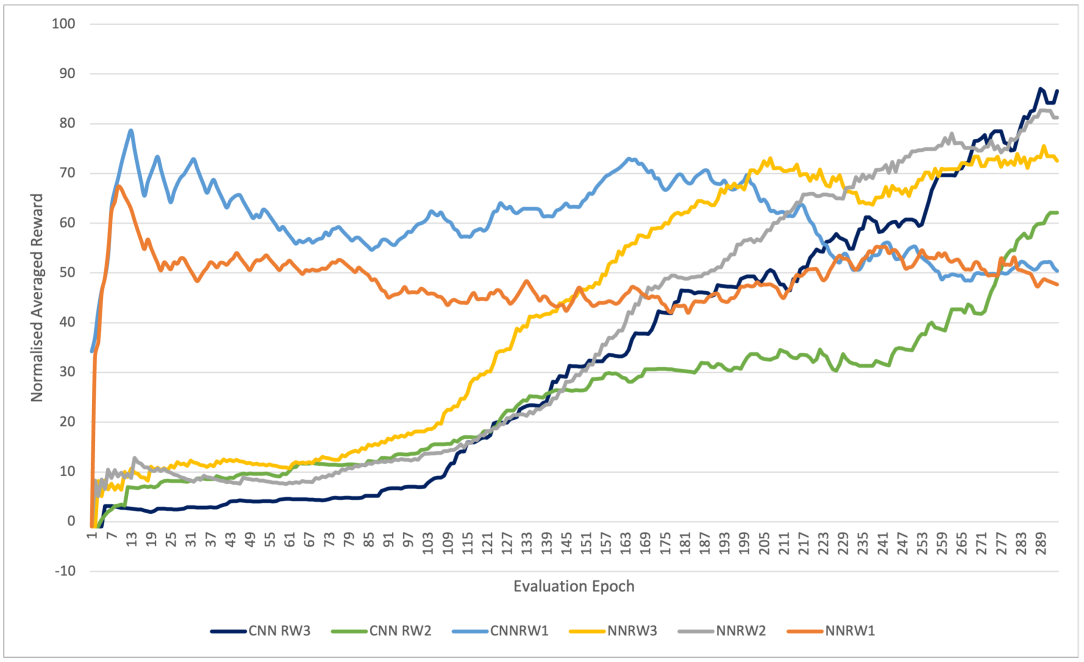

FIGURE 8.5: Comparison of average rewards on an evaluation epoch in function of the epoch for the different DQN controllers (from Excel file)

8.1.2 Hypothesis n°2

When looking at the comparison between the lap time, average minimal LiDAR distance and average acceleration/deceleration metrics on the RedBull track (Figures 8.1, 8.2, 8.3 and 8.4), we can draw 3 conclusions regarding Hypothesis n°2 (*CNNs offer better results over selected metrics than NNs for autonomous racing on the F1Tenth platform using data from a UST-10LX LiDAR.*):

- Regarding the speed metric (time to complete a full lap), Hypothesis n°2 is verified: for each reward function, the CNN controller performs better than the NN controller.
- Regarding the safety metric (average minimum LiDAR distance), Hypothesis n°2 is also verified: for each reward function, the CNN controller performs better than the NN controller.
- Regarding the smoothness metric (average acceleration/deceleration), Hypothesis n°2 is verified for reward functions 1 and 2.

Hypothesis n°2 was then checked using the P-Value method, similarly to Subsection 8.1.1, using the following hypotheses:

- **Null Hypothesis (H0)**: There is no real advantage provided by CNNs over NNs using the same reward function.
- **Alternative hypothesis (H1)**: There is a real advantage provided by CNNs over NNs using the same reward function.

Using the standard value of Alpha = 0.05 and an independent two-tailed T-test, the results for all metrics were found to be significant; more details are available in the Excel file at https://github.com/HL-Boisvert/MSc_Project_HW.
To conclude, Hypothesis n°2 is verified for all metrics and reward functions apart from reward function 3 for the smoothness metric.

8.1.3 Hypothesis n°3

Hypothesis n°3 (*DRL-based controllers can generalise to new race tracks without affecting too much the metrics*) was verified by comparing on the Monaco track the performance of the CNNs trained on the RedBull track to those trained on the Monaco track.
The P-values were then calculated, still using the standard value of Alpha = 0.05 and an independent two-tailed T-test (Table 8.1).

Metric	Reward Function 2	Reward function 3
Time (s)	0.99>0.05	0.64>0.05
Average Positive Acceleration (m/s^2)	0.74>0.05	0.98>0.05
Average Negative Acceleration (m/s^2)	0.77>0.05	0.34>0.05
Average Minimal LiDAR distance (m)	0.91>0.05	0.93>0.05

TABLE 8.1: P-Values for all metrics on Monaco Track when comparing DQN controllers trained on Monaco or RedBull tracks

All P-Values are larger than 0.05, meaning there is a significant statistical difference between the controllers trained on the Monaco track and those trained on the RedBull track.

The metric are compared in Table 8.2 and in Figures 8.6, 8.8, 8.9, 8.7: The Monaco-trained controllers perform better on the speed metric by around 10%, however, the RedBull trained controllers perform better on the safety metric by around 5%. The performance is similar on the smoothness metric.

Furthermore, we can also see that even the CNNs trained on a different track perform better than the Wall Following controller on all metrics.

To conclude, Hypothesis n°3 is considered verified as there is no major difference in performance on the Monaco track for controllers trained on Monaco or a different track. Those results show that CNNs are able to generalise to new tracks without significant performance changes.

Controller	Total Time (s)	Average Positive Acceleration (m/s^2)	Average Negative Acceleration (m/s^2)	Maximum Acceleration (m/s^2)	Maximum Deceleration (m/s^2)	Average Minimum LiDAR Range (m)
CNN RW2	+10.55%	+3.23%	-3.20%	+1.16%	-1.35%	+2.50%
CNN RW3	+9.95%	+5.95%	-3.51%	0.92%	-9.29%	+7.23%

TABLE 8.2: Comparison of metric changes on the Monaco track when going from Monaco trained controllers to RedBull trained controllers

8.1.4 Hypothesis n°4

Concerning Hypothesis n°4 (*The sim2Real gap is negligible (minor changes of the metrics) and doesn't affect the performance of the controllers implemented.*), we first looked at the Wall Following controller. Once the PID was tuned, there was no significant statistical difference between the simulator and the physical car, with $p-value = 0.019 < 0.05$ for the speed metric and $p - value = 0.039 < 0.05$ for the safety metric.

Looking at the CNN with reward function n°3, there was also no significant statistical difference, with $p-value = 0.046 < 0.05$ for the speed metric and $p-value = 0.033 < 0.05$ for the safety metric.

Table 8.3 shows that the only significant (>5%) difference between the simulator and

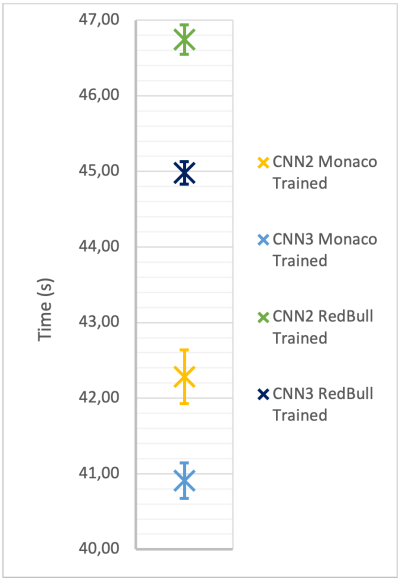

FIGURE 8.6: Lap time comparison on the Monaco track (from Excel file)

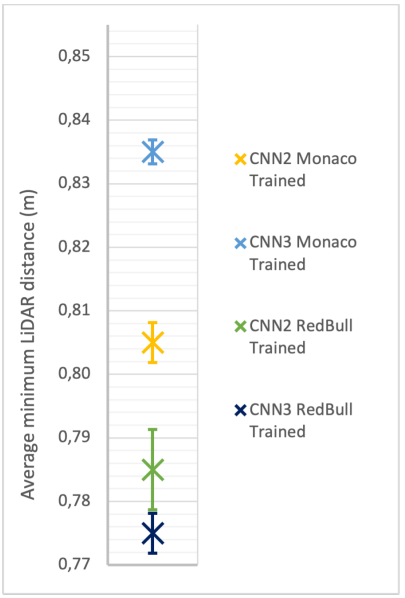

FIGURE 8.7: Average minimal LiDAR distance on the Monaco track (from Excel file)

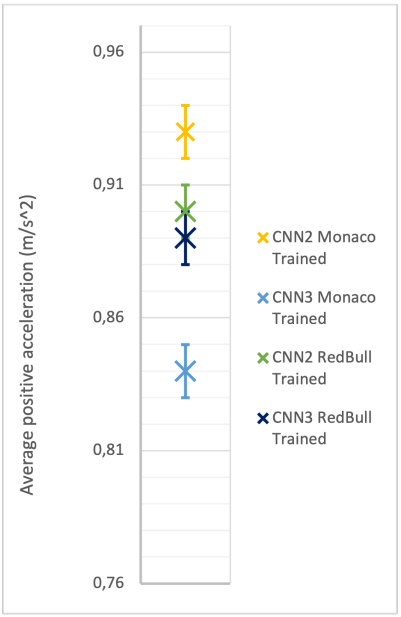

FIGURE 8.8: Average acceleration comparison on the Monaco track (from Excel file)

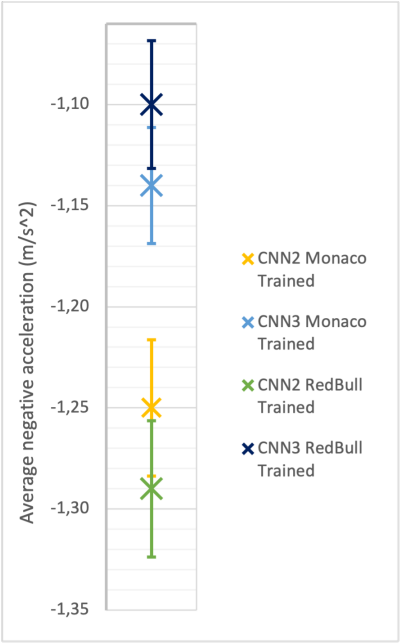

FIGURE 8.9: Average deceleration comparison on the Monaco track (from Excel file)

the physical car is the 20% time increase for the CNN controller using reward function n°3.

Compared to manual driving, we can conclude that both those controllers are performing better regarding both the speed and safety metrics, with the DQN controller performing the best.

To conclude, Hypothesis n°4 cannot be considered verified as there is a significant increase in the time to complete a full lap on the CNN with reward function n°3.

Controller	Time (s)	Average Minimum LiDAR range (m)
Wall Following	+0.52%	-2.56%
CNN RW3	+20.39%	-0.55%

TABLE 8.3: Comparison of metric changes when going from the simulator to the physical car

There are four limitations to Hypothesis n°4:

1. The lack of the speed input because of the unreliable odometry, which means the metrics that depended on the speed measurement couldn't be assessed
2. The lack of data for a complete lap of the track with the DQN controller
3. The lack of data at speeds higher than 4m/s
4. The lack of data for DQN controllers other than the CNN with reward function n°3

8.2 General Conclusions

We can now make some general conclusions from this project:

- DRL has proven to be quite time-consuming compared to Wall Following, especially the training; however, this is a hardware limitation, and the training process could be improved by fine-tuning the hyper-parameters.

- Choosing an appropriate reward function is critical: with an inappropriate function, the DQN controller cannot complete a full lap on the Monaco and Silverstone tracks.
- When using an appropriate reward function (n°2 and 3), DQN performs better than Wall Following on all metrics, even on never-seen-before tracks (H1).
- CNNs provide a real advantage over NNs according to all metrics (H2).
- With our parameters, DQN can generalise to new tracks (H3), although not perfectly (compared to a controller specifically trained for the track).
- There is a non-negligible Sim2Real gap both for Wall Following and DQN. Regarding Wall Following, once the PID is tuned again, the controller performs similarly to the simulator. Regarding DQN, the controller had an issue preventing it from completing a full lap. However, when extrapolating, the DQN controller still performs better than Wall Following and Manual driving according to the speed and safety metrics. The Sim2Real gap is visible when comparing the performance of DQN on the physical car to the one in the simulator, especially regarding the speed metric (H4).
- Our results have several limitations, mainly due to time constraints, which could be an area of improvement in future works, listed in Section 8.3.

8.3 Future works

This dissertation introduced a well-defined protocol in Chapter 6; because this experimental protocol is quite flexible, the process could be extended to other DRL state-of-the-art methods, such as DDPG and TD3, following the same methodology.
A first improvement to the experimental protocol would be to fine-tune the different DQN hyper-parameters: learning rate α, discount factor γ, greediness *epsilon*, epsilon decay, and size of the replay buffer. This would make sure that the training process is as efficient as possible.
A second improvement would be to train the controllers for longer to then test their limits; this couldn't be done due to hardware and time constraints.
A third improvement to the experimental protocol would be verifying hypothesis n°3

more rigorously. The protocol used here to verify this hypothesis was to train the DQN controller on the RedBull racetrack before testing it on the Silverstone and Monaco tracks. The underlying assumption was that those three tracks offered a comparable level of challenge, but this hasn't been proven. Because the underlying assumption may be false, it makes the assessment unreliable. A way to improve this would be by defining a metric to quantify the difficulty offered by a specific race track based on the length, minimal track width and minimum curve radius. This could be implemented using a Support Vector Machine (SVM) with the shape of the track as an input, as introduced in Ploeg [2015].

A fourth improvement to the experimental protocol would be to develop a more advanced simulator to perform the training, for example, by using a more accurate kinematic model rather than the Bicycle model we used. A fifth and last improvement would be to use the car's speed as an input for the DQN controllers, which could improve the performance regarding the speed metric.

Appendix A

Maps

Those three race tracks will be used to evaluate the performance of the controllers. They have been chosen because they present different types of challenges, with combinations of long straight lines and sharp corners.

A.1 Formula 1 Track - Red Bull Ring

The race track for training the controllers is based on the Red Bull Ring race track, located in Austria and hosting the Austrian Grand Prix. the `.png` file that will be used in ROS (Figure A.1) is based on Wikipedia [2022a].

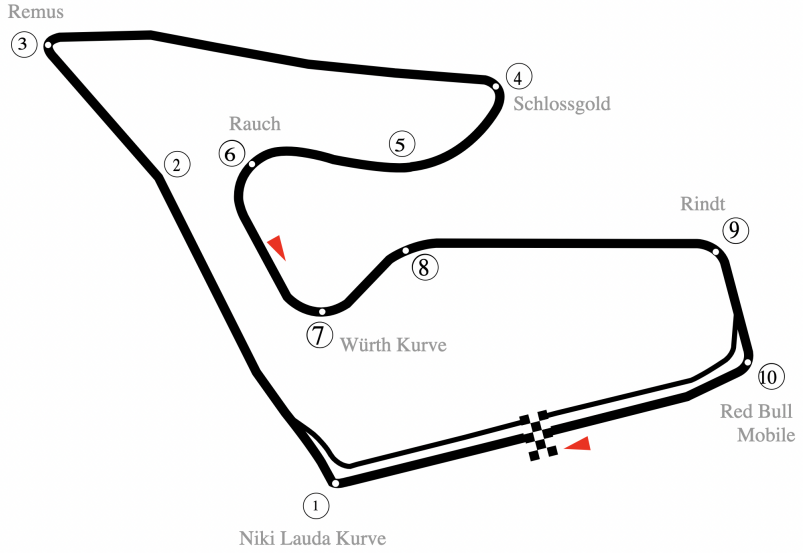

FIGURE A.1: Red Bull Ring Racing Track template from Wikipedia [2022a]

A.2 Formula 1 Track - Silvertone Circuit

The Silverstone circuit is located in England and hosts the British Grand Prix; the `.png` file that will be used in ROS (Figure A.2) is based on Wikipedia [2022b].

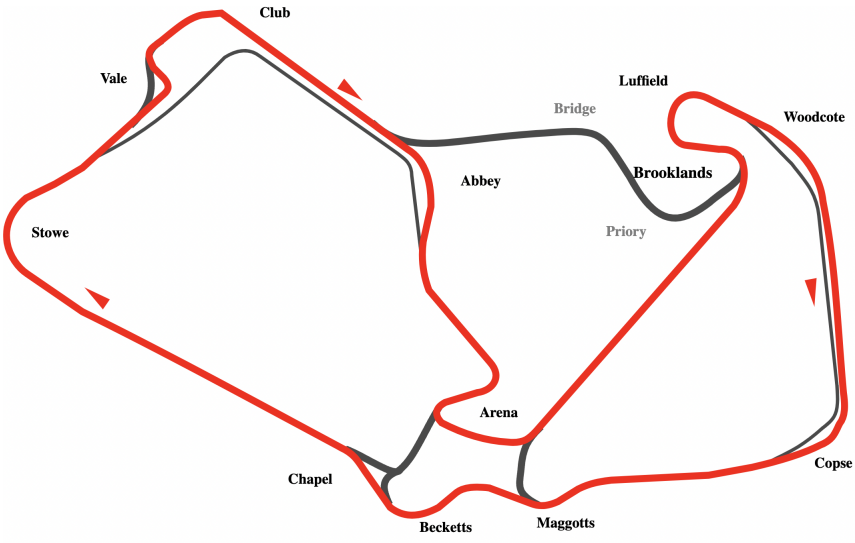

FIGURE A.2: Silverstone Circuit template from Wikipedia [2022b]

A.3 Formula 1 Track - Circuit de Monaco

The Circuit de Monaco is located in Monaco and hosts the Monaco Grand Prix; the `.png` file that will be used in ROS (Figure A.3) is based on Wikipedia [2022c].

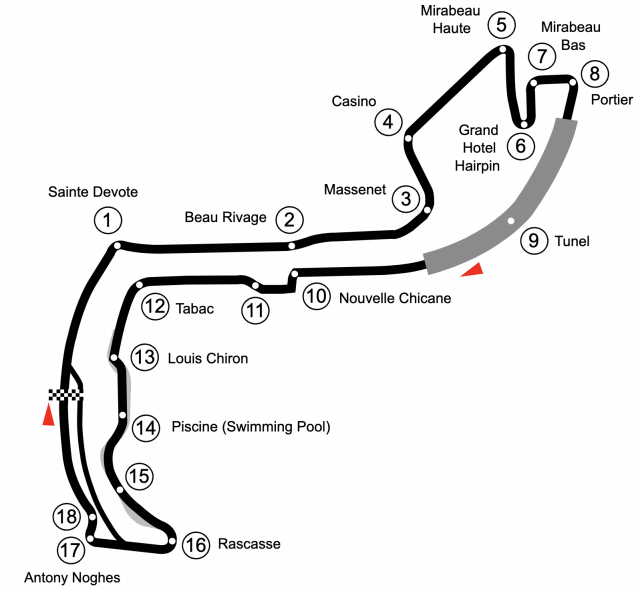

FIGURE A.3: Circuit de Monaco template from Wikipedia [2022c]

A.4 Oval Track

This oval race track was designed by myself and is used to evaluate the sim-to-real gap.

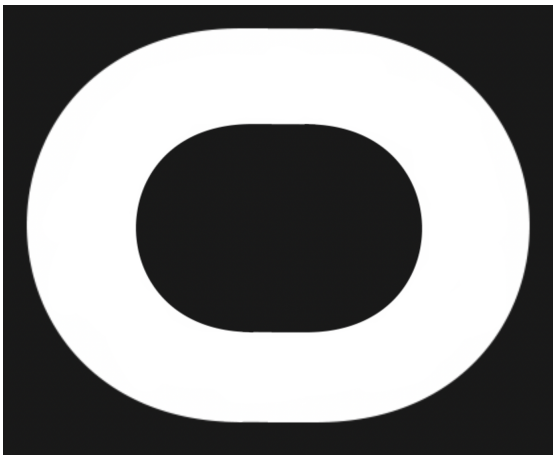

FIGURE A.4: Oval-shaped racetrack, inspired by the Mesa Marine Raceway

Bibliography

Ahmad El Sallab, Mohammed Abdou, E. P. S. Y. (2017). Deep reinforcement learning framework for autonomous driving. pages 70–76.

Alloghani, M., Al-Jumeily, D., Mustafina, J., Hussain, A., and Aljaaf, A. J. (2020). *A Systematic Review on Supervised and Unsupervised Machine Learning Algorithms for Data Science*, pages 3–21. Springer International Publishing, Cham.

Balaji, B., Mallya, S., Genc, S., Gupta, S., Dirac, L., Khare, V., Roy, G., Sun, T., Tao, Y., Townsend, B., Calleja, E., Muralidhara, S., and Karuppasamy, D. (2020). Deepracer: Autonomous racing platform for experimentation with sim2real reinforcement learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2746–2754.

Barabás, I., Todoruţ, A., Cordoş, N., and Molea, A. (2017). Current challenges in autonomous driving. *IOP Conference Series: Materials Science and Engineering*, 252:012096.

Bosello, M., Tse, R., and Pau, G. (2022). Train in austria, race in montecarlo: Generalized rl for cross-track f1tenth lidar-based races. In *2022 IEEE 19th Annual Consumer Communications Networking Conference (CCNC)*, pages 290–298.

Brunnbauer, A., Berducci, L., Brandstätter, A., Lechner, M., Hasani, R., Rus, D., and Grosu, R. (2021). Model-based versus model-free deep reinforcement learning for autonomous racing cars.

Brunnbauer, A., Berducci, L., Brandstätter, A., Lechner, M., Hasani, R., Rus, D., and Grosu, R. (2022). Latent imagination facilitates zero-shot transfer in autonomous racing.

Fuchs, F., Song, Y., Kaufmann, E., Scaramuzza, D., and Durr, P. (2021). Super-human performance in gran turismo sport using deep reinforcement learning. *IEEE Robotics and Automation Letters*, 6(3):4257–4264.

Garcia, F. and Rachelson, E. (2013). *Markov Decision Processes*, chapter 1, pages 1–38. John Wiley and Sons, Ltd.

Gerrish, S. (2018). *How Smart Machines Think*.

Hansson, S. O., Belin, M. A., and Lundgren, B. (2021). Self-driving vehicles—an ethical overview. *Philosophy and Technology*, 34.

Hermansdorfer, L., Betz, J., and Lienkamp, M. (2020). Benchmarking of a software stack for autonomous racing against a professional human race driver.

Ivanov, R., Carpenter, T., Weimer, J., Alur, R., Pappas, G., and Lee, I. (2020). Case study: verifying the safety of an autonomous racing car with a neural network controller. pages 1–7.

Li, Y. and Ibanez-Guzman, J. (2020). Lidar for autonomous driving: The principles, challenges, and trends for automotive lidar and perception systems. *IEEE Signal Processing Magazine*, 37(4):50–61.

Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *CoRR*.

Mangharam, R. (2020). F1tenth autonomous racing course. https://linklab-uva.github.io/autonomousracing/assets/files/assgn4_2021.pdf/. Accessed: 07/04/2022.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529–33.

Navarro, A., Genc, S., Rangarajan, P., Khalil, R., Goberville, N., Rojas, J., and Asher, Z. (2020). Using reinforcement learning and simulation to develop autonomous vehicle control strategies.

O'Kelly, M., Sukhil, V., Abbas, H., Harkins, J., Kao, C., Pant, Y. V., Mangharam, R., Agarwal, D., Behl, M., Burgio, P., and Bertogna, M. (2019). F1/10: An open-source autonomous cyber-physical platform.

(ORBIT). Orbit area 4p framework. https://www.orbit-rri.org/about/area4pframework/. Accessed: 06/04/2022.

O'Shea, K. and Nash, R. (2015). An introduction to convolutional neural networks. *ArXiv e-prints*.

(PEP8). Pep8. https://peps.python.org/pep-0008/. Accessed: 05/04/2022.

Ploeg, R. v. d. (2015). Modeling race track difficulty in racing games.

Popescu, M.-C., Balas, V., Perescu-Popescu, L., and Mastorakis, N. (2009). Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8.

(PyStyleGuide). Pystyleguide. http://wiki.ros.org/PyStyleGuide. Accessed: 05/04/2022.

Rivera, D. E., Morari, M., and Skogestad, S. (1986). Internal model control: Pid controller design. *Industrial & engineering chemistry process design and development*, 25(1):252–265.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms.

Sejnowski, T. J. (2018). *The deep learning revolution*. MIT press.

Silver, D. (2015). Lectures on reinforcement learning. URL: https://www.davidsilver.uk/teaching/. Accessed: 04/03/2022.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T. P., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

Tsitsiklis, J. and Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690.

Watkins, C. and Dayan, P. (1992). Technical note: Q-learning. *Machine Learning*, 8:279–292.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. PhD thesis, King's College.

Weber, T., Racanière, S., Reichert, D., Buesing, L., Guez, A., Jimenez Rezende, D., Badia, A., Vinyals, O., Heess, N., Li, Y., Pascanu, R., Battaglia, P., Silver, D., and Wierstra, D. (2017). Imagination-augmented agents for deep reinforcement learning.

Wikipedia (2022a). Wikipedia article. https://en.wikipedia.org/wiki/Red_Bull_Ring. Accessed: 07/04/2022.

Wikipedia (2022b). Wikipedia article. https://en.wikipedia.org/wiki/Silverstone_Circuit. Accessed: 07/04/2022.

Wikipedia (2022c). Wikipedia article. https://en.wikipedia.org/wiki/Circuit_de_Monaco. Accessed: 07/04/2022.