Autonomous racing on unseen tracks using reinforcement learning

by
Devin Jefferies

Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering (Electronic) in the Faculty of Engineering at Stellenbosch University.

Supervisor: Dr J. C. Schoeman
Co-Supervisor: Dr B. D. Evans

March 2025

Acknowledgements

I would like to take this opportunity to acknowledge the people and organisations that contributed to my Master's degree:

- The Council for Scientific and Industrial Research (CSIR) for funding my Master's degree.
- The Autonomous Vehicle Systems (AVS) lab at the Technical University of Munich for hosting me for my research exchange and allowing me the opportunity to test my algorithms on their vehicle and track.
- My two supervisors for their time and guidance during this project.
- The other members of the Electronic Systems Lab who have gone through this journey with me and helped make my time in the lab memorable.
- My family and friends for their support and motivation throughout my Master's degree.

Stellenbosch UNIVERSITY IYUNIVESITHI UNIVERSITEIT
forward together
sonke siya phambili
saam vorentoe

Plagiaatverklaring / *Plagiarism Declaration*

1. Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
   Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.

2. Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
   I agree that plagiarism is a punishable offence because it constitutes theft.

3. Ek verstaan ook dat direkte vertalings plagiaat is.
   I also understand that direct translations are plagiarism.

4. Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
   Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism

5. Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
   I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

D.W.P. Jefferies	**01/11/2024**
Voorletters en van / *Initials and surname*	Datum / *Date*

Abstract

The rise of autonomous systems in the vehicle industry has highlighted their potential to increase safety and convenience, transforming the way vehicles interact with their environment. The racing domain is used to further explore the capabilities of autonomous systems, as it serves as an ideal test bed to push these algorithms to their performance and safety limits. In the racing domain, autonomous racing algorithms are tasked with generating control commands (such as speed and steering angle) to navigate a vehicle around a track as safely and as quickly as possible. In order to achieve this goal, autonomous racing algorithms that employ classical control methods can be used. These methods rely on accurate track and vehicle models to race according to preplanned trajectories around racetracks. This allows for consistent and repeatable racing behaviour; however, it limits their use to known, static environments. In comparison, deep reinforcement learning algorithms can learn to race without the need for these preplanned trajectories. These algorithms learn from a trial-and-error process, making them more applicable to generalisation. This generalisation ability makes them an alternative to classical algorithms in unseen and changing environments that are more reminiscent of real-world conditions. However, RL algorithms do have limitations, as they tend to underperform in comparison to classic methods.

In this thesis, we introduce an end-to-end racing framework with improved performance that is comparable to classic algorithms while increasing its ability to generalise to unseen tracks. Our method uses a centre-orientated twin delayed deep deterministic policy gradient (CO-TD3) agent to race on the standard F1TENTH platform. We input sensor measurements into a deep reinforcement learning network and teach an agent how to race by controlling a vehicle's speed and steering angle. We illustrate the effects that an optimal agent state vector and reward function have on racing performance and generalisation ability. In addition, we present a random track generator that can be used for research and testing various algorithms in the F1TENTH simulator. To illustrate the performance of our CO-TD3 agent, we conduct experiments in simulation and the results are compared to a benchmark of current racing algorithms. Our algorithm demonstrates robustness and generalisation ability by racing on a real vehicle after being trained in a simulated environment. These results demonstrate that our CO-TD3 agents are capable of achieving performance comparable to classic control algorithms in simulation, while also generalising effectively to unseen tracks both in simulation and in the real world.

Uittreksel

Die toename van outonome stelsels in die voertuigbedryf het hul potensiaal beklemtoon om veiligheid en gerief te verhoog, wat die manier waarop voertuie funksioneer en met hul omgewing saamwerk, verander. Een van die opwindende maniere om verder ondersoek in te stel oor die vermoëns van outonomiese stelsels is outonomiese wedrenne. Wedrenne bied ‘n unieke en uitdagende omgeving wat die limiete en grense van outonomiese stelsels verder uitbrei. Hierdie outonome wedrenalgoritmes het die taak om beheeropdragte (soos spoed en stuurhoek) te genereer om ’n voertuig so veilig en so vinnig moontlik om ’n baan te navigeer. Om dit te bereik, gebruik outonome wedrenalgoritmes wat klassieke beheermetodes gebruik, akkurate baan- en voertuigmodelle om op optimale trajekte om renbane te jaag. Dit maak voorsiening vir konsekwente en herhaalbare wedrengedrag; dit beperk hul gebruik egter tot bekende, statiese omgewings. In vergelyking, kan diep versterking leer algoritmes leer om te jaag sonder die behoefte aan hierdie modelle. Hierdie algoritmes leer uit ’n proef-en-fout-proses, wat hulle meer toepaslik maak op veralgemening. Hierdie veralgemeningsvermoë maak hulle ’n alternatief vir klassieke algoritmes in ongesiene en veranderende omgewings wat meer aan werklike toestande herinner. Versterking leer algoritmes het egter beperkings, aangesien hulle geneig is om te onderpresteer in vergelyking met klassieke metodes.

In hierdie tesis stel ons ’n end-tot-end-renraamwerk bekend met verbeterde werkverrigting wat vergelykbaar is met klassieke algoritmes, terwyl ons die vermoë daarvan om te veralgemeen na ongesiene bane verhoog. Ons metode gebruik ’n middelpunt-georiënteerde dubbel vertraagde diep deterministiese beleidgradiënt (CO-TD3) agent om op die standaard F1TENTH-platform te jaag. Ons voer sensormetings in ’n diep versterkingsleer netwerk in en leer ’n agent hoe om te jaag deur ’n voertuig se spoed en stuurhoek te beheer. Ons illustreer die effekte wat ’n optimale agenttoestandvektor en beloningsfunksie op wedrenprestasie en veralgemeningsvermoë het. Daarbenewens bied ons ’n lukrake baangenerator aan wat gebruik kan word vir navorsing en toetsing van verskeie algoritmes in die F1TENTH simulator. Om die prestasie van ons CO-TD3 agent te illustreer, word dit in simulasie geëvalueer en die resultate word vergelyk met ’n maatstaf van huidige wedrenalgoritmes. Ons algoritme demonstreer robuustheid en veralgemeningsvermoë deur op ’n regte voertuig te jaag nadat dit in ’n gesimuleerde omgewing opgelei is. Dit wys dus dat ons CO-TD3 agente in staat is om prestasie te verrig wat vergelykbaar is met klassieke beheeralgoritmes, terwyl hulle in staat is om te veralgemeen na ongesiene bane in simulasie en in die werklike wêreld.

Contents

List of Figures

List of Tables

Nomenclature

Notation

$\mathbb{E}$	expectation
$\mathbb{I}$	indicator function
∇_θ	gradient with respect to parameter θ
$\leftarrow$	assignment
x	scalar
$\lvert x \rvert$	absolute value of variable x
$\overline{x}$	maximum value of scalar x
$\underline{x}$	minimum value of scalar x
$\mathbf{x}$	vector
$\mathbf{X}$	matrix
$\mathcal{X}$	set

Acronyms and abbreviations

DNN	deep neural network
RL	reinforcement learning
DRL	deep reinforcement learning
DQN	deep Q-Network
DDPG	deep deterministic policy gradient
TD3	twin delayed deep deterministic policy gradient
CO-TD3	center-oriented twin delayed deep deterministic policy gradient
sim-to-real	simulation-to-reality
MPC	model predictive control
LVP	linear parameter varying
SGP	simplified Gaussian process
MLMS	meta learning-based multi-scenario
SAC	soft actor-critic
A3C	asynchronous actor-critic
IL	imitation learning
LMPC	learning model predictive control
TAL	trajectory-aided learning
MDP	Markov decision process
POMDP	partially observed Markov decision process
TD	temporal difference
FOV	field of view
ROS	Robot Operating System
SLAM	simultaneous localisation and mapping
PF	particle filter
CDDT	compressed directional distance transform
MPCC	model predictive contouring control
DO	Dreamer + Occupancy

Chapter 1

Introduction

Autonomous systems have seen significant advances in recent years, proving effective in controlled environments across various industries. However, their application in complex real-world scenarios, such as autonomous racing, presents unique challenges that remain largely unsolved. Human racers possess the ability to interpret visual input and make decisions based on past experiences, even if they have never experienced that exact situation before. Replicating this capability is one of the biggest challenges faced with fully autonomous racing algorithms, as it is nearly impossible to predict and plan for every possible scenario in an ever-evolving environment such as the real world. An autonomous racer's ability to generalise its actions to a set of unseen features is crucial to its ability to perform in these complex environments. Achieving this ability will allow autonomous racers to successfully navigate unseen tracks with performance comparable to that of an autonomous racer on a seen track. Additionally, generalising will enable the autonomous racer to perform in dynamic environments, a crucial aspect in multi-vehicle racing. Having an autonomous racer with these capabilities will greatly benefit the field of autonomous racing, highlighting the potential and efficacy of these autonomous racers in unpredictable real-world conditions.

The increased interest in this field has led to the emergence of many autonomous racing leagues, such as the Abu Dhabi Autonomous Racing League [1], Indy Autonomous Racing Challenge [2] and the F1TENTH league [3]. For this project, the F1TENTH platform will be used. This platform is commonly used among universities to test autonomous racing algorithms, facilitating direct comparisons between algorithms developed on this platform. Furthermore, it offers a simulated and a physical platform, which allows algorithms to be developed safely using the simulated environment and then tested on the real vehicle, allowing a direct comparison between simulated and real-world tests.

1.1 Research motivation

The primary objective of autonomous racing algorithms is to safely and quickly navigate a vehicle around a racetrack, a task made significantly more challenging on unseen tracks. Although classic control algorithms have been applied to address this, they tend to perform

well only in known conditions, often struggling with changing environments and unknown scenarios, which limits their broader applicability.

Furthermore, this task becomes increasingly difficult as the complexity of the environment grows and uncertainties arise about how performance in simulated environments will transfer to real-world conditions. The complexity of these challenges highlights the need for methods that can reliably operate in new environments without additional tuning between deployments. Additionally, the racing environment serves as an ideal platform for developing these systems as they compel autonomous systems to operate at the performance edge. The complexity of this challenge fosters creative innovation, which has historically driven advancements in the commercial automotive sector. Developing autonomous systems that excel at the performance edge not only advances the field of autonomous racing but also lays the groundwork for future innovations in commercial autonomous vehicles.

1.2 Aims and objectives

In response to the challenges posed by complex and unpredictable real-world environments, this project aims to develop a deep reinforcement learning (DRL) autonomous racing agent capable of reliable performance on unseen tracks. The objectives of this agent can be identified as follows:

- Increasing the performance and reliability of current DRL autonomous racing algorithms on seen track.
- Achieving the ability to generalise racing behaviour to unseen tracks.
- Increasing the robustness of these algorithms to allow for seamless simulation-to-reality transfers.

If successful, the development of this agent aims to enable autonomous racing in real-world environments. Additionally, our agent should operate with performance comparable to that in simulation without additional training.

1.3 Solution overview

Classical racing algorithms have been able to achieve consistent racing performance using different frameworks such as model predictive control [4] and trajectory optimisation with tracking [5]. These classic frameworks, which use a model of the track and vehicle, are able to achieve good racing performance on seen tracks due to their ability to plan an optimal trajectory. All of these algorithms have one hurdle that prevents them from performing on unseen tracks, which is their reliance on track models and predefined plans. Furthermore, these methods are more susceptible to model mismatch when transferring from a simulated

environment to a real environment as they rely on the accuracy of simulated vehicle models for real world performance. These limitations highlight the need for algorithms that can be used to achieve more adaptive performance and negate the effects of model mismatch.

Deep reinforcement learning (DRL) end-to-end algorithms have emerged as a possible alternative to classic control methods [6]. Through trial and error, deep reinforcement learning agents can develop a robust policy that can generalise well to unseen situations [7]. This eliminates the need for extensive modelling and allows these algorithms to adapt to diverse environments. These algorithms do have downsides, as they can have unpredictable and inconsistent behaviour [8, 9]. Furthermore, tests using these RL algorithms are generally performed in simulation, not on physical vehicles [6]. However, addressing these challenges would enable the deployment of these autonomous racing systems in unseen environments with the robustness and adaptability needed for real-world success.

To address the limitations of classical algorithms and improve the adaptability of deep reinforcement learning methods, we introduce a new method to achieve end-to-end racing on the standard F1TENTH platform [3]. This DRL method uses the twin delayed deep-deterministic policy gradient algorithm (TD3) to control the vehicle using processed LiDAR sensor data, resulting in a centre-orientated TD3 (CO-TD3) racing agent. By developing a general racing strategy, we improve the limitations of current DRL formulations by increasing the repeatability and reliability of these methods. This implementation highlights strategies that can be used to increase the generalisation ability of RL racing agents. This includes optimising the training of these agents to expose them to features of a racetrack in such a way as to allow for the development of a robust and comprehensive racing policy.

Our method shows that including a well-constructed agent state vector, comprising of sensor measurement and a centring reference term, increased the consistency of the agent's action selection and consequently improved the overall performance of the agent. This centring term is calculated from the data captured by the LiDAR scan and provides the agent with an indication of how far it is from the centre of the track at each time step. This term is used by the agent to better position itself on the track. Lastly, a reward function is designed that incorporates aspects such as the centring term and the lap time of the agent to reward the agent for making safe and fast progression around the track. All of this combined to create an agent that has increased performance compared to classic and other DRL methods. Additionally, this method was able to easily transfer from a simulated environment, where it was trained, to hardware, where it was able to perform as expected. Thus, we overcome the obstacle of not only traditional sim-to-real transfer but additionally sim-to-real transfer in an unseen environment.

1.4 Thesis outline

This thesis presents a structured approach to using Deep Reinforcement Learning (DRL) for autonomous racing. Specifically, the work investigates how DRL can provide a robust solution to the autonomous racing problem by allowing agents to navigate previously unseen tracks and perform effectively in both simulated and real world settings.

A comprehensive review of the literature in Chapter 2 examines the progression from traditional control strategies in autonomous racing to modern DRL techniques. The discussion highlights notable performance achieved using both classic control and DRL methods and concludes with an evaluation based on the use of these methods to solve the autonomous racing problem. Chapter 3 delves into the theoretical background required to understand the methodologies used in this research, covering reinforcement learning concepts such as Markov decision processes (MDPs), deep neural networks (DNNs), and the twin delayed deep deterministic policy gradient algorithm (TD3).

In Chapter 4, the racing problem is formulated and discussed. This chapter also presents specific information about the simulated environment used in this thesis. Chapter 5 focuses on the design and structure of our CO-TD3 agent's state and action vectors, the reward function, as well as the tuning of reward function weights and network hyperparameters.

Chapter 6 outlines a training strategy that focusses on the development of a robust racing policy. This approach enables the identification of the maximum speed at which the algorithm can effectively control the vehicle.

Chapter 7 shows the impact the training track has on the agent's generalisation ability, as well as describes the creation of a random track generator that is used to increase the number of tracks in the set and create tracks similar to those expected in the real world.

Chapter 8 addresses the sim-to-real transfer challenge, exploring the gap between simulated training and real-world deployment. The effectiveness of noise modelling in LiDAR scans, as well as the robustness of agent behaviours to unseen physical conditions, is discussed in relation to bridging this gap.

Chapter 9 examines the racing performance achieved by these our CO-TD3 agent both in simulation and on physical hardware. Finally in Chapter 10 we provide a conclusion and a discussion of possible future work.

Chapter 2

Literature review

The field of autonomous racing uses many different methods and algorithms in the pursuit of fast and safe racing behaviour. These approaches can be broadly categorised into three methods: classic, end-to-end, and partial-end-to-end. This section provides a brief overview of these methods and distinguishes them by their implementation of the autonomous driving pipeline. Key algorithms from each category are introduced, first by giving a brief technical overview and then by providing a review of their application in autonomous racing. This is done to investigate the potential generalisation ability of each algorithm and to identify how aspects of these algorithms can be used in the creation of an autonomous racing algorithm that can race on unseen tracks, in both simulation and in the real world. These algorithms are then evaluated according to the objective described in Chapter 1 and the methods and techniques required by each algorithm.

2.1 Classic autonomous racing methods

The classic autonomous racing approaches focus on using traditional control algorithms. These methods rely on mapping the track to generate an optimal path around the track. The goal of these algorithms is to follow this path by assessing its position on the track relative to the optimal preplanned position. Based on its position on the track, the algorithm issues control commands based on rules and constraints incorporated into the algorithm that aim to keep the vehicle on this optimal path. Furthermore, it is possible to incorporate ideal speed profiles for the vehicle to match along its trajectory [6]. These aspects of classic methods allow for consistent and competitive performance. However, classic methods are more suited to static environments, as the predefined plan usually does not account for environmental changes.

The classic autonomous vehicle pipeline is used to differentiate the subsystems responsible for different tasks within the algorithm. Therefore, a brief overview of the pipeline will be discussed before addressing technical information and use cases for various algorithms like model predictive control, trajectory optimisation and tracking, and follow-the-gap.

2.1.1 Autonomous driving pipeline

Classical methods operate using a commonly referred to classic driving pipeline [6]. This pipeline encompasses the subsystems that enable the vehicle to go from inputting environmental data from a sensor to calculating control commands to navigate the vehicle. The three subsystems (perception, planning, and control) can be optimised and arranged in various ways depending on the requirements specified by the algorithm used. This allows for variations in how data flows between subsystems and how each subsystem is optimised, depending on the algorithm's requirements. Figure 2.1 shows one of these implementations.

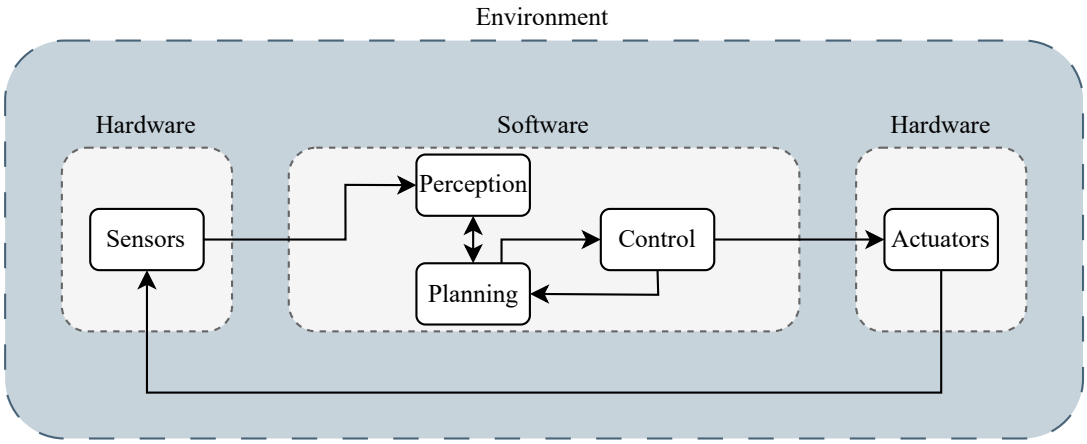


Figure 2.1: A possible implementation of the autonomous driving pipeline where the black arrows show the flow of information along the pipeline. The feedback arrow shows that the output of the pipeline makes the vehicle change its position in the environment and thus a new vehicle state can be identified by the sensors which starts the process again

Perception

Perception refers to the mechanism that gathers information about the environment. This is done with sensors that detect information about elements of the environment or of the vehicle. Perception systems typically involve localisation, which determines the vehicle's position within its environment, object detection to identify obstacles or other vehicles, boundary detection to recognise lane markings or road edges, and mapping to create a detailed representation of the surrounding area, enabling it to make informed decisions [10]. It is an integral aspect of the system, as it forms an important connection from the software to the information available from the environment. Perception provides crucial information for other elements of the pipeline and allows autonomous vehicles to execute plans based on their current state in the environment [10].

Planning

A common approach to planning involves a 3-level hierarchical structure, comprising route planning, behavioural planning, and motion planning, each addressing different aspects of navigation [11]. The 3-level hierarchical approach is shown in Figure 2.2 [12]. This subsystem is key in dynamic and unseen environments as it aims to react to environmental elements.

The route planner's goal is to execute a plan based on the path that it's required to take. This planner is more orientated towards navigating to a way point rather than the specific behaviours of the system [10, 12].

A behavioural planner aims to achieve local objectives by interacting with elements of the environment and following a specific set of rules that dictate the limit of its behaviour [11]. This planner handles immediate changes such as obstacle avoidance or lane changes. [10, 12].

A motion planner is used to generate trajectories or sequences of actions to navigate to local objectives. This should aim to find the optimal path for the current state. The solutions to these motion planning problems are often complex and require immense computing power; therefore, numerical approximations are typically used [12].

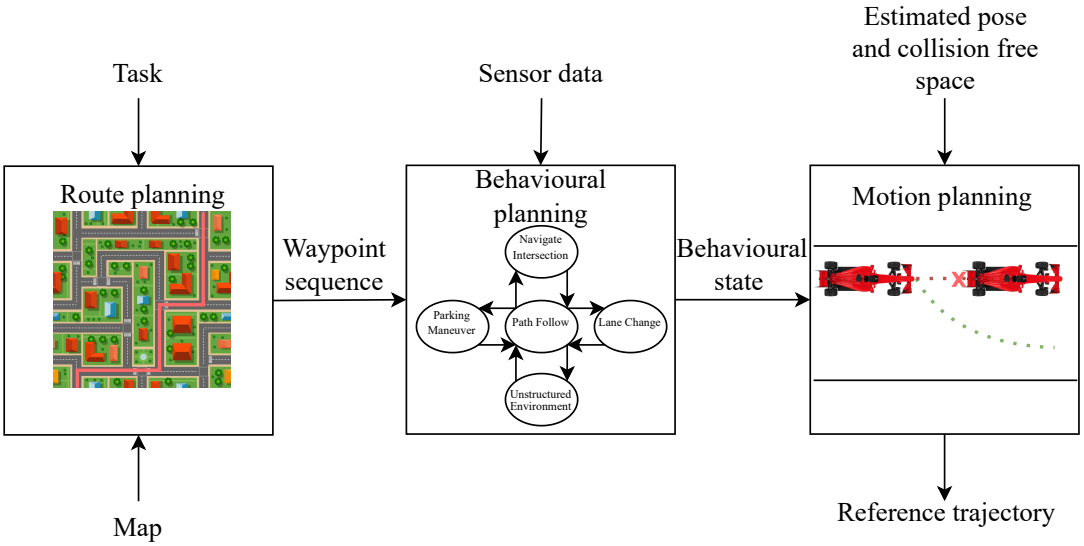


Figure 2.2: The hierarchy of the autonomous driving planning process adapted from Pandel et al. [12]

Control

This subsystem focuses on keeping the vehicle on the planned trajectory. The controller works in conjunction with the planner by translating high-level trajectories into executable actions. This involves having a controller with effective feedback mechanisms that aim to reduce tracking errors. Feedback mechanisms are crucial, as they allow the system to

detect and correct deviations from the planned path, ensuring that the vehicle remains on course despite external disturbances or inaccuracies. The controller processes this information to generate specific actions, such as adjusting the steering angle and speed, which propel the vehicle along the predefined trajectory.

2.1.2 Model predictive control

Model predictive control (MPC) is a control strategy that makes use of the classic autonomous pipeline. MPC predicts the vehicle's future trajectory for a finite future time window called the horizon. It bases this prediction on the vehicle's current parameters and system constraints. These constraints can be path boundaries, safety requirements, or vehicle dynamics. The computation of an optimal sequence of states is done by solving an optimisation problem [4]. Once this sequence of states is determined, the first control action is executed. This happens repeatedly on the next time step with the information from the updated state.

MPC is often referred to as 'receding horizon' control as the algorithm attempts to achieve long-term optimality using short-time optimisation. The combined use of optimisation and prediction is the feature that differentiates this algorithm from conventional control methods [13]. As MPC aims at finding control inputs by solving an optimisation problem that minimises the cost function rather than just following a pre-planned trajectory, it can be used in more dynamic environments.

Figure 2.3 shows an MPC at time K with a prediction horizon of N_3. The MPC aims to follow the reference by issuing control commands to get the control variable to match the reference.

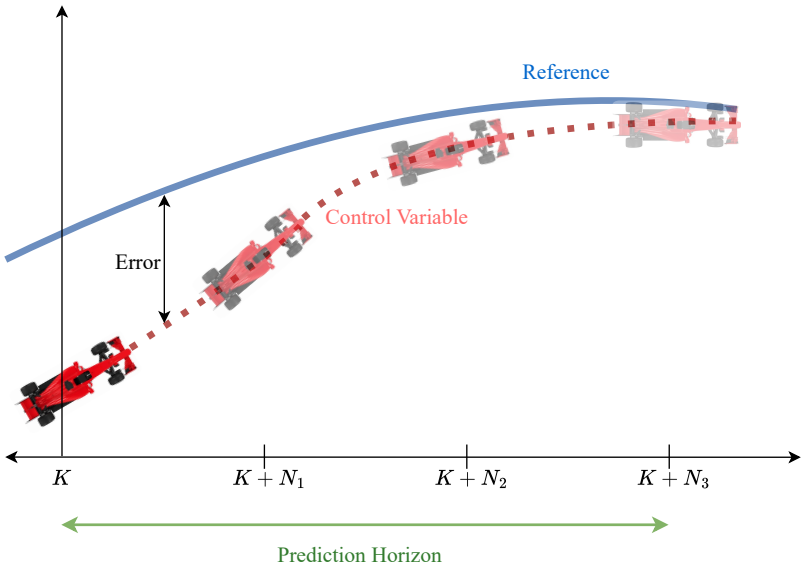


Figure 2.3: MPC predictions with horizons N_1, N_2, N_3 with each prediction getting closer to the reference trajectory

It issues steering and speed commands to attempt to achieve the reference trajectory and speed profile. It shows that the predictions on the horizon are closer to the reference as the MPC algorithm reduces the error.

Brown and Gerdes [4] created a nonlinear MPC to handle complex racing situations. These authors make use of mathematical encoding and accurate models to quickly compute control inputs. Their implementation was able to achieve lane switching, on a real vehicle, caused by a pop-up obstacle. This shows some generalisation ability by reacting to dynamic scenarios. However, this deviation is only for a limited period, and it returns to the original planned path after the deviation. The authors note that the controller has to solve a non-convex optimisation problem whenever it is faced with an obstacle, and convergence is not guaranteed for this problem. Therefore, this controller would not be able to handle a completely unseen environment as the lack of certainty in convergence and the computational cost of continually solving this optimisation problem will hinder its ability to perform. Lastly, the vehicle model used to transfer from the simulated to the real environment was highly accurate, which makes the system more susceptible to model mismatch if there are ever discrepancies between the simulated and real versions. However, Brown and Gerdes [4] ability to achieve dynamic response on a real vehicle shows the potential adaptability of MPC to unseen environments.

Alcala et al. [14] used a linear parameter varying (LPV) MPC that aims to reduce the computational cost of MPC algorithms, which was a major hurdle in preventing performance in unseen environments. The F1TENTH platform was used to test their MPC algorithm and the vehicle was modelled using the standard bicycle model [15]. This model is used to simplify the vehicle model by creating a two-wheeled vehicle model similar to a bicycle. They tested their implementation in simulation and found that the vehicle was able to follow a reference trajectory and complete laps on a mapped track. The authors did note that they were unable to eliminate the steady-state error in the longitudinal direction due to slow convergence. They perform the same tests on hardware and achieve successful laps comparable to their simulated results. They do find limitations brought on by the localisation error. The system was not robust enough to handle errors in the sensors used for the localisation, even with the addition of a Kalman filter. These authors show that MPC is capable of racing in both software and hardware. Their attempt to lower the computational cost of MPC addresses one of the main limitations preventing performance on unseen tracks. However, the results that the authors achieved are still limited to seen tracks.

Yue et al. [16] developed a simplified Gaussian process (SGP) that reduces the complexity of the residual model. This is an attempt to reduce the computational burden of complex optimisation problems. Furthermore, these authors developed a novel meta-learning-based multi-scenario model (MLMS). This model updates the model online to increase its ability to adapt and generalise to unseen scenarios. A gradient descent function

is used to adjust the weight of the online parameters to achieve this. The generalisation ability is tested using vehicle masses that differ from the initial model. The controller was able to update the model parameters to account for these discrepancies. They also reported that their SGP reduced overall computational cost. This work shows that there are ways to mitigate the computational cost of traditional MPC and this can be done to increase generalisation; however, this generalisation is focused towards the model and not the environment.

MPC has shown to be a viable option for autonomous racing; however, the computational cost involved seems to limit its uses to seen environments where it can handle only slight deviations like unexpected obstacles. The computational cost of continuously solving optimisation problems needs to be overcome if it is to be implemented in a completely unseen environment.

2.1.3 Trajectory optimisation and tracking

Trajectory optimisation is a method of extracting an optimal trajectory that minimises lap time around a race track. This is used in conjunction with a pure pursuit controller that enables the vehicle to follow this generated path. This controller attempts to stay on the path by reducing the error between the reference trajectory and the current position. It uses localisation to generate a reference position for the vehicle based on sensor input usually in the form of a LiDAR scan. The algorithm then uses a 'lookahead point' on the preplanned path as the target to navigate towards. Consequently, the steering angle is computed to 'pursue' this lookahead point [5].

This pure pursuit controller finds a curvature path from its current position to the lookahead point. Figure 2.4 shows the curved path, with curvature derived from R, from its current position to the goal.

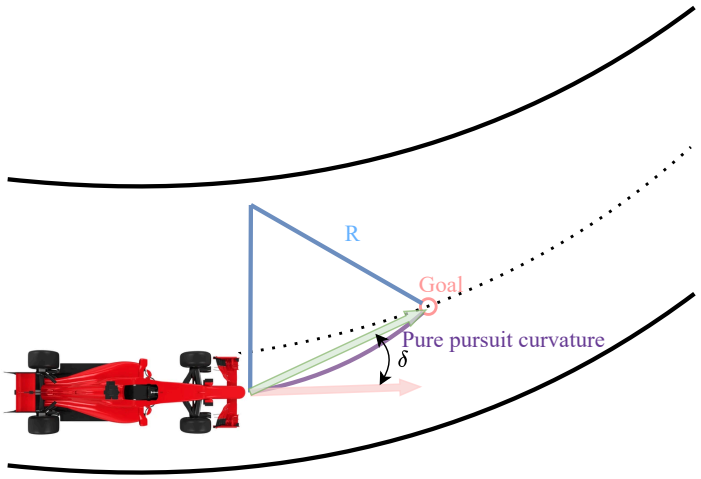


Figure 2.4: Calculating the desired path using a pure pursuit algorithm adapted from Sukhil and Behl [5].

Geometric constraints are added to the driving equation to create Ackermann steering. This enables the calculation of a steering angle δ to guide the vehicle along the curvature [5].

An important aspect of the pure pursuit controller is the lookahead distance. An optimal value of the lookahead is crucial for the fastest lap time around the track. It also influences its ability to navigate track features. When the lookahead value is too small, it can cause unwanted oscillations in the vehicle's motion. In contrast, if the value is too large, it can cause the controller to miss features in the track causing it to collide with the boundary as seen in Figure 2.5.

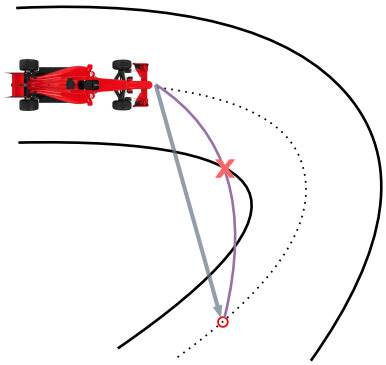


Figure 2.5: Non-optimal lookahead value that causes a collision with the track boundary as the goal point is too far ahead and the curvature path fails to account for the track boundary between the vehicle and the goal.

Kapania and Gerdes [17] designed a feedback-feedforward steering controller. The controller aims to minimise lateral path deviation while operating at the steering limits. The incorporation of steering feedforward better estimates a steering angle for a known velocity profile and curvature path. This should reduce tracking error as the compensation required by the steering feedback is minimised. Additionally, they included side slip information in the feedforward portion of the controller to remove the system's dependence on real-time side slip feedback. This created a system with robust stability. The development of this algorithm was to test if it would be effective as a real world control algorithm. This was proven as the system was successfully deployed on a full-scale Audi TTS. However, the authors did note that the system is limited by its sensitivity to model uncertainty, which is more prominent at the physical limits of handling. This shows the feasibility of these methods to operate in the real world.

Sukhil and Behl [5] proposed an adaptive lookahead algorithm to change this lookahead value based on the upcoming track geometry. With long straights, the lookahead distance should be greater, as it allows for higher speeds, but in sharp turns, the value should decrease to avoid the vehicle cutting corners and colliding with the track boundaries. They define three driving objectives, a maximum speed, a minimum deviation, and a convex combination of the previous two objectives. The convex combination uses a trade-off factor

to prioritise one of the objectives. These three methods were tested using the same path around a track. The convex combination outperformed the other methods by achieving faster lap times and higher average speeds. Furthermore, they managed to transfer their algorithm to the F1TENTH hardware and achieve similar results to their simulated tests, as shown in Table 2.1.

Environment	**Lap Time [s]**	**Average Speed [m/s]**
Simulation	9.33	2.097
Real-world	9.44	2.042

Table 2.1: Results from tests achieved by Sukhil and Behl [5] showing these authors ability to achieve comparable results between simulation and hardware tests

This shows that the author's implementation is robust to sim-to-real transfer; however, this implementation was tested in ideal and static environments where localisation error and track uncertainties were not hindering factors. In addition, the track has to be mapped to generate the trajectory. Therefore, these methods do seem to have the robustness required for real-world use; however, their reliance on a predetermined plan really hinders the ability to operate in unseen environments.

Becker et al. [18] introduce a lateral geometric controller combined with a non-linear Pacejka tyre model [19]. The tyre model is used to model the interaction between the vehicle tyre and the ground. This creates a hybrid method that combines elements of classic geometric methods and model-based methods that they call MAP. This method aims to increase performance by leveraging the simplicity of geometric control and the accuracy of model-based methods. The authors compared three versions of controllers to establish a baseline for their methods' performance in simulation. MAP with Pacejka outperformed MAP with a linear tyre model and a regular pure pursuit controller by achieving the lowest tracking error and deviation while achieving a maximum speed of 11 m/s. This method is also tested on hardware where a maximum speed of 8 m/s was achieved. Becker et al. [18] show how accurate geometric controls can aid in performance and lower the computational cost of traditional MPC controllers. Although this is still limited to known and static environments, advances in these two methods could aim to lower the controller complexity and computational cost to incorporate more reactive control methods that are adaptive to changing environments.

The performance and simplicity of pure pursuit controllers make them attractive for high-speed racing. However, accurate localisation limits its use to environments where reliable localisation can be achieved. Therefore, it is currently not plausible to deploy them in dynamic or unseen environments. However, the possibility of using aspects of pure pursuit controllers to simplify more complicated control algorithms such as MPC

could lead to their use in more complex environments.

2.1.4 Follow-the-gap

Follow-the-gap is an obstacle avoidance algorithm that surveys the current environment and plans a trajectory based on the observation. Various aspects are used to determine the optimal path. The main mechanism that dictates the trajectory of the vehicle is the gap between the obstacles. The size of the gap and the distance to the obstacles dictate the required heading to manoeuvre through the most optimal gap [20].

The algorithm follows a continuous three-step process. First, the maximum gap is identified in the gap array. Next, the angle to the centre of the gap is then calculated. Finally, the heading angle is determined. This allows the algorithm to navigate the vehicle through diverse and dynamic environments, and no prior information about the environment is required. This contradicts many classic algorithms that first generate a plan and then execute it, as this algorithm generates a plan based on the current state of the environment. This also negates the reliance on accurate localisation methods and the need for track and vehicle models. Figure 2.6 shows how a LiDAR would scan the environment and identify the largest gap through which the vehicle can fit. The blue arrow shows the heading that the vehicle should follow to pass through the centre of the gap.

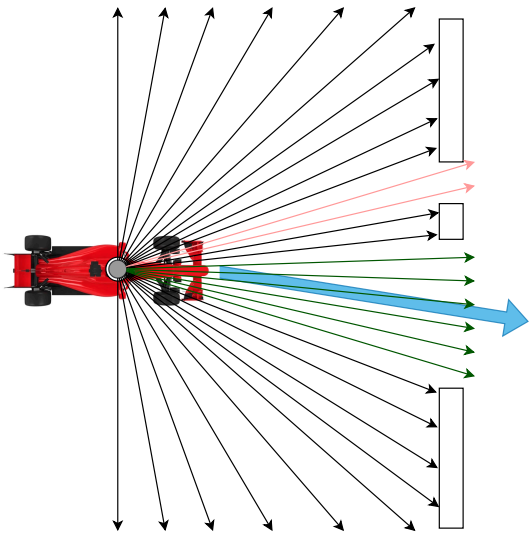


Figure 2.6: An overview of the LiDAR scan for a follow-the-gap algorithm with the green beams representing a viable path and the red beams representing a gap that is too small. The blue arrow represents a viable path that the algorithm has identified.

Demir and Sezer [21] used a follow-the-gap method to plan overtaking manoeuvres. They specified a two-vehicle problem where the ego vehicle's goal is to overtake the actor vehicle. The algorithm uses a goal point in the front of the actor vehicle. The trajectory

to this point accounts for obstacles like the actor vehicle and the boundaries. The goal point is also used as a reference point to prompt the ego vehicle to merge back into the correct lane. They tested their implementation in simulation and found it to be 13% safer and 41% more comfortable than other overtaking algorithms.

Hossain et al. [22] propose a follow-the-gap method with a dynamic window to determine control commands for mobile robots. This proposed solution aims to generate collision-free trajectories with moving obstacles. They showed that their method was able to generate a goal-orientated collision avoidance trajectory for low-speed robots in dynamic environments. They also showed that their algorithm works on hardware by testing it on a TurtleBot Robot [23]. These experiments showed their method's ability to generate smooth and safe trajectories in the presence of static and dynamic obstacles at slow speeds.

The follow-the-gap method shows promise as it is robust to environmental changes. Its lack of dependence on previous track knowledge and highly accurate vehicle models makes it ideal for racing on unseen tracks. Despite this, the algorithm is more suited to slower velocities as the lack of optimal path planning and vehicle models limits this algorithm. Its lack of foresight, sensitivity to sensor noise, and inability to handle high-speed cornering make it less suitable for racing and, therefore, it is unlikely that it will be able to compete in a racing environment.

2.2 End-to-end autonomous racing methods

As classic control methods are more suited to known environments, other methods with the capability to generalise need to be considered. Methods that have the potential for this are end-to-end methods. These racing methods replace the 3 components of the classic pipeline with a neural network. As seen in Figure 2.7 a neural network is now responsible for the perception, planning, and control of the autonomous vehicle.

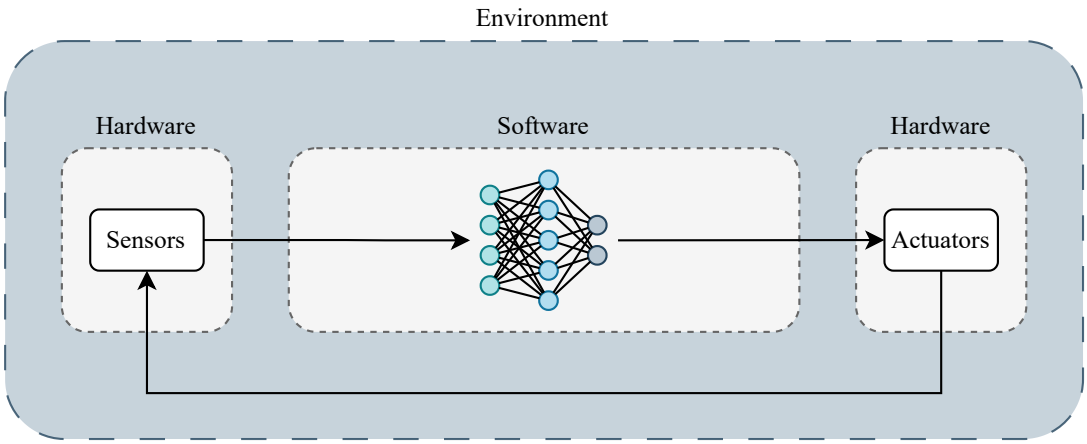


Figure 2.7: End-to-end autonomous driving pipeline

2.2.1 Reinforcement learning

Reinforcement learning has emerged as a popular method for robotic control [24–26]. The structure of an RL network consists of layers of artificial neurons that form a connection between the input and output of the network. It learns by updating these neurons by using experiences and trial and error in the environment. The learning process is guided by a reward function that quantifies the quality of certain actions by assigning it a numeric value. The RL agent aims to maximise this value by taking actions that it considers to be the most optimal [27].

Applications of racing using reinforcement learning vary widely based on the type of input data and algorithms used. Researchers have investigated using RL in the racing domain by applying it to different tasks. For example, Cai et al. [28] used a soft actor-critic (SAC) to achieve autonomous drifting instead of racing. These authors showed that the RL-based drift controller can drift on tracks that it had not seen in training; however, a reference trajectory is provided. This showed robustness to different vehicle types and friction coefficients. These authors showed the benefits of RL by having a controller with the ability to generalise, whereas a traditional classic controller would not be able to perform in these conditions.

RL has also been used in racing with visual data as the input [29, 30]. Jaritz et al. [29] used an asynchronous actor-critic (A3C) algorithm with image input to control a vehicle in a realistic rally game. They attain results that prove the generalisation ability of this algorithm on an unseen track. However, it was only able to do so in simulation with ideal inputs generated for the game. Similarly, Fuchs et al. [31] were able to achieve racing performance that outperformed human drivers in the video game Gran Turismo. The authors use a soft-actor-critic (SAC) algorithm to achieve these results. The agent's state space consists of the velocity, acceleration, the previous steering command, a binary crash indicator, the Euler angle (the angle of horizontal vehicle rotation from the tangent at the current centerline point), distance measurement (from the vehicle centre to its surrounding objects and track boundaries with a 180° field-of-view) and a sample of N upcoming centerline curvatures. The network used this to select actions that consisted of a steering angle and a throttle-break signal. This is used with an exponentially discounted future reward signal based on progress along the track. This is accompanied by a crash penalty that is scaled relative to the car's kinetic energy on impact.

Using this implementation, the authors achieved lap times quicker than the game's built-in AI and 52,303 human drivers. To test the robustness of their agent, it was transferred to a different car from the one it was trained on and tested on the same track it was trained on. Furthermore, it was tested on a different track than it was trained on while using the same car as in training. Both of these cases showed the generalisation performance of their agent as it achieved a faster lap time than the built-in AI and humans

once again.

The authors did note limitations of this generalisation where in difficult track features like hairpin turns, the agent was not able to extrapolate its behaviour to these unseen scenarios. This shows that end-to-end methods are highly effective and can generalise better than any classic method. The development of end-to-end methods for racing games does have limitations if it were to be transferred into the real-world. These limitations include the reliance on difficult-to-access information that is used in the state space such as the Euler angles and the upcoming curvature of the track. These would require detailed previous knowledge of the real track and highly accurate localisation if it were to be recreated in the real world. With this limitation, its ability to race on hardware is unknown. Therefore, if the end goal is to race on unseen tracks in simulation and in the real world, there cannot be a reliance on information that is only available in simulation. Despite this, their implementation was successful and aspects of it can be adjusted to fit a domain more closely orientated to the real world.

In contrast, Stachowicz et al. [32] had a hardware-forward approach. These authors used a camera-based system to achieve a high-speed driving policy that can be deployed in a variety of diverse indoor and outdoor environments. The network underwent offline training before being trained via the input from the vehicle camera. The offline training was done by training the agent on a large-scale dataset with navigation trajectories from many existing environments. This data focused on low-speed driving, which is not the desired behaviour. Consequently, aggressive, high-speed behaviour must be developed in real-world online training. This method enabled the agent to learn a real-time driving policy on hardware. This implementation shows the robustness and transferability of end-to-end methods by achieving results from a combination of simulation and real world learning. This highlights the usability of these methods in unseen real-world scenarios.

To simplify the racing problem, Bosello et al. [33] discretised the action space. This allows for the use of simpler algorithms, such as a deep Q-network (DQN). Using this simpler action space, Bosello et al. [33] achieve consistent racing performance and show that this method has some generalisation capabilities by racing it on unseen tracks. However, to achieve racing performance comparable to that of human drivers, these methods must work in a continuous action space. Their discretised action space does not allow for precise control over the vehicle, which is necessary in high-speed racing. These authors note that the next advancement in their research is to address this limitation. Although this limitation exists, their implementation aims to address the generalisation problems in racing and therefore can be used as a baseline comparison.

LiDAR as an input to an RL racing algorithm on the F1TENTH platform has been shown to be a viable option [34–36]. Ivanov et al. [35] compared the results of the deep deterministic policy gradient (DDPG) algorithm with the twin delayed deep deterministic policy gradient (TD3) algorithm. The authors also reported on how the number of beams

in the observation space affects the performance. This showed that the TD3 algorithm used with 21 beams resulted in a good performance that showed robustness to the sim-to-real transfer. However, they reported that their solution was not robust enough to overcome any LiDAR faults on the real vehicle.

Notably, Brunnbauer et al. [34] showed that a model-based deep-RL algorithm using LiDAR data can generalise to unseen tracks. Although their methods performed well, it was unable to achieve 100% completion on seen or unseen tracks consistently. The authors achieved results, on a real vehicle, by successfully driving around a seen track and showed some generalisation performance by driving the track in the reverse direction, although no success rate was reported. Once again, this research can serve as a comparison for future results.

RL end-to-end methods address many limitations of classic methods and are more suited to the general behaviour required to race on unseen tracks. Although it is still very rooted in simulation, the advancements of these algorithms have led to more instances of researchers addressing the sim-to-real problem. The robustness and generalisation ability of these algorithms contribute to their ability to bridge the sim-to-real gap, as there is less of a reliance on accurate vehicle and track models.

2.2.2 Imitation learning

Another technique that complements RL by enhancing learning efficiency is imitation learning (IL). IL is a machine learning method where an IL agent learns by mimicking the behaviour of an expert. The agent learns from a dataset that contains demonstrations that the expert has generated. Through this, the agent attempts to recreate the behaviour in this dataset [37].

Pan et al. [38] used IL to implement agile off-road autonomous driving. They aimed to map high-dimensional sensor data to continuous speed and steering commands. They used an MPC as the expert with a combined batch and online algorithm. They tested their algorithm on a 1/5th scale rally car on an outdoor dirt track. They aimed to show the benefit of an IL agent trained on MPC data rather than just using an MPC as the learner can operate from image data. The authors showed that IL with online training performed better than batch training. Furthermore, the online agent achieved speeds similar to the MPC expert on the dirt track, proving the effectiveness of their algorithm.

Zhang and Cho [39] aim to address the unexpected behaviour that is usually encountered when using IL by selecting examples from different sources. The authors alternate between collecting from both trained policies and reference policies. This allows for the primary policy to correct its path which results in better vehicle states. The authors implementation of a safety classifier is an extension of a previous IL algorithm called DAgger. This implementation reduces the number of queries to a reference policy. This was tested

in a simulated environment called TORCS. The authors showed that their algorithm outperformed DAgger by producing a faster and safer racer. Their method also allowed the primary policy to mimic the reference policy more closely.

2.3 Partial end-to-end autonomous racing methods

Partial end-to-end methods offer a middle ground, combining the strengths of both classic and end-to-end approaches. These methods typically involve the integration of learnt components within a traditional pipeline, enabling the system to take advantage of the precision of classic methods while still benefiting from the adaptability of machine learning techniques. This hybrid approach allows for a more structured learning process while still providing the flexibility needed to handle dynamic and unseen environments. Figure 2.8 shows an example of a partial end-to-end pipeline. This implementation is commonly used with the MPC algorithm to create a learning MPC (LMPC) [40–42] where model parameters or trajectories are learnt rather than being explicitly defined.

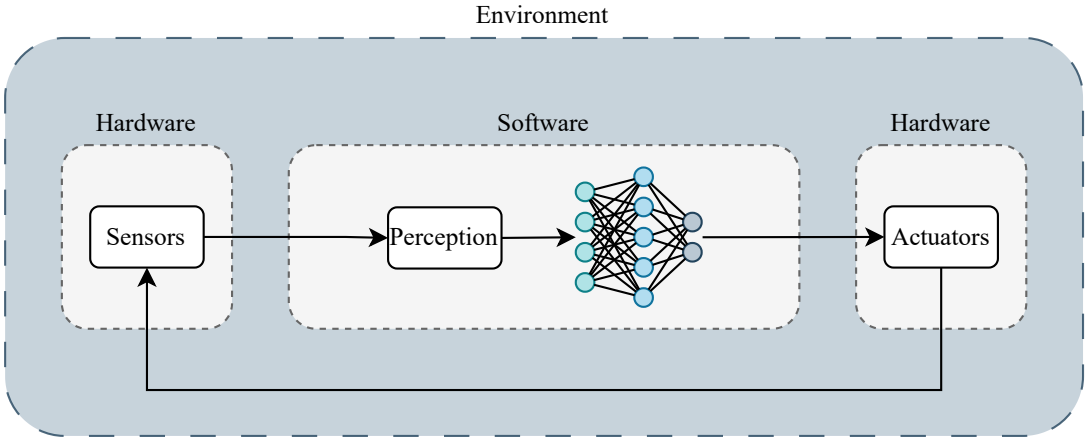


Figure 2.8: An example of a partial end-to-end autonomous driving pipeline where the planning and control aspects are done using a neural network

Ghignone et al. [43] present a trajectory-conditioned RL controller (TC-Driver) that aims to use RL to increase the robustness of autonomous racing controllers. Their framework uses a classic planner and an RL agent for the control. They used the capabilities of RL models to heuristically deal with model mismatch and track generalisation while using reliable classic planning methods. They focus on using RL to model lateral tyre forces as that is difficult to model in high-speed racing. Their implementation was tested in simulations and was able to outperform traditional MPC in the presence of model mismatch by lowering the crash rate by 23.81%. However, the MPC was able to achieve much faster lap times. They also achieved track generalisation with their method achieving a crash rate of 12.7% on portions of a track that their algorithm has not been exposed to,

although these were done in ideal situations when no model mismatch was present. This result shows that it improves the generalisation as compared to classic methods without any learning elements. However, it is not able to reliably generalise its racing behaviour.

Evans et al. [36] used trajectory-aided learning (TAL) to increase the reliability of RL racing agents. TAL uses an end-to-end framework but incorporates aspects from classical methods such as providing the agent with the racing line and speed profile. The racing line provides the agent with an optimised trajectory to minimise lap time, and the speed profile gives the optimal speeds along this racing line. This information enables the agent to have better positioning and action selection on the track. Therefore, it is not in the pipeline but the use of a classic path planner in addition to the end-to-end framework distinguishes it from other implementations. The agent receives more reward if its trajectory is similar to that of the pre-calculated trajectory. This is shown to increase the performance of end-to-end agents by increasing the completion rate on seen and unseen tracks. Although the authors showed that TAL was better than their baseline end-to-end implementation, they were still unable to reliably achieve 100% completion on seen or unseen tracks.

2.4 Evaluation of methods

Each framework used for autonomous racing provides insight into how we can achieve robustness and reliability for autonomous racing. However, some methods are more suited to racing on unseen tracks than others. Classic algorithm's requirement for track models severely limits their use in dynamic or unseen environments. This also significantly affects these algorithms' ability to bridge the sim-to-real gap. Classic methods require that the environment and models are nearly identical to what is used in simulation, and any deviation from this can cause a decrease in performance. The incorporation of neural networks into the classic frameworks aims to solve these frameworks' reliance on these models and addresses the decrease in performance seen from model mismatch. However, these models still rely on some trajectory planning or classic control, making them unsuitable for unseen tracks. Full end-to-end DRL algorithms negate the need for any prior knowledge of the track as they can generalise their actions to perform on unseen tracks. DRL relies on the training phase to prepare the agent to perform in the environment. The aspects of the training phase and the DRL agent can be adjusted to increase its generalisation performance. These DRL methods are less affected by the sim-to-real transfer as they do not rely on accurate models or localisation to perform. This is highly beneficial in real world environments, as it will not hinder the agent's ability to select actions. All of these make DRL methods ideal for racing competitively on unseen tracks and effectively bridging the sim-to-real gap. Table 2.2 shows a collection of popular autonomous driving algorithms along with the prerequisites and limitations of each algorithm.

Algorithm	**Instances of Use**	**Prerequisites**	**Common Limitations**
Trajectory optimisation & tracking	[5, 17, 18, 44–49]	Track and vehicle model, Trajectory optimisation algorithm	Model accuracy, Lookahead distance optimisation, Localisation
MPC (Model Predictive Control)	[4, 14, 50–59]	Accurate vehicle and track model	Localisation, Model mismatch
Follow the Gap	[20–22]	Vehicle dimensions	Poor performance in complex environments
End-to-end RL (Reinforcement Learning)	[28–30, 32–35, 60–62]	Training phase, simulation environment	Sensor data quality, long training time
End-to-end IL (Imitation Learning)	[38, 39]	Training phase, Expert demonstration data	Quality of expert data
Partial end-to-end (combination of IL and traditional methods)	[36, 40–43, 63, 64]	Some model, Training phase	Model accuracy, Localisation, performance variance

Table 2.2: Comparison of different algorithms

Chapter 3

Preliminaries

Vehicle control using deep reinforcement learning (DRL) introduces unique challenges that differentiate it from classical control methods. One of the primary difficulties is the inherent lack of interpretability within DRL models. Unlike traditional approaches, where control logic and decisions are clearly defined, RL agents operate through a policy learnt from the data, making it difficult to understand the reasoning behind their actions. This black-box nature of neural networks limits the ability to track the agent's decision-making process, making it challenging to diagnose performance issues or identify areas where improvements are needed. Understanding the technical details of the RL algorithms becomes essential to guide the agent's behaviour and ensure it meets the intended performance criteria. This chapter provides the necessary preliminary knowledge of DRL and the TD3 algorithm, highlighting the key concepts required to address the unique challenges of using it for vehicle control.

3.1 Reinforcement learning

Reinforcement learning describes the process of learning optimal behaviours through trial and error. In an RL problem, the goal is typically to maximise the agent's cumulative reward over time by selecting actions that achieve desirable outcomes. Depending on the algorithm, this may involve learning an optimal policy π that maps a state s to a possible action a or learning a value function that predicts the future reward of actions taken from different states. Some RL methods aim to directly learn this mapping (as in policy-based methods), while others focus on evaluating action values (as in value-based methods). Formally, RL operates within the context of a Markov Decision Process (MDP), where the future is independent of the past, given the present state. This property, known as the Markov property, ensures that the decisions made by the agent depend solely on the current state, which makes it unnecessary to reference previous states [7, 65]. A game of chess is a useful analogy for this concept. At any point in the match, the best move can be determined solely by the current position of the pieces, without needing to know the sequence of moves that led there [7]. [1]

[1] There are chess exceptions that should be known and accounted for such as castling or en passant (capturing a pawn on a double move)

In an MDP, the agent interacts with the environment, observes its state s_t at time t, takes action a_t according to its policy $\pi(a_t|s_t)$, and receives a reward r_t based on the outcome of the action. The environment then transitions to a new state s_{t+1} according to the transition probability $p(s_{t+1}|s_t, a_t)$ as shown in the flow diagram Figure 3.1. The value of the reward r_t received by the agent is dictated by the reward function. This function is designed to encourage preferred behaviour and discourage unfavoured behaviour. It is important that the reward function effectively represents the desired behaviour, as this is the main mechanism that guides the agent's performance. The agent does not actually know what task it is supposed to complete, its only goal is to maximise its rewards. Therefore, the reward function must ensure that the agent will only receive the maximum reward when performing the task optimally.

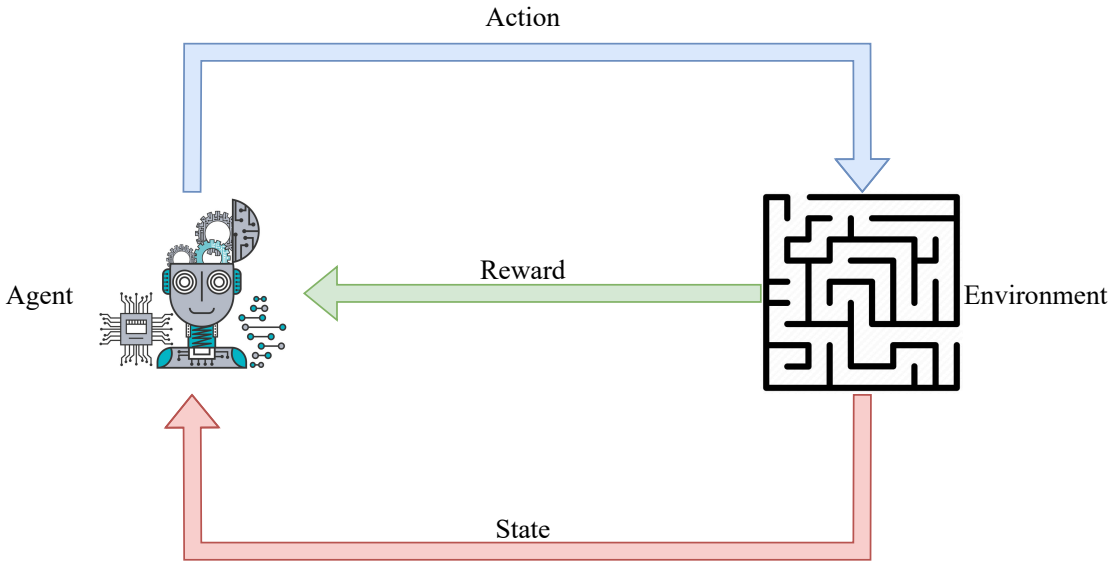


Figure 3.1: Reinforcement learning agent's interaction with the environment where an action a_t is executed in the maze environment. This action produced a reward r_t based on the quality of the action and a new state s_{t+1} that the agent can use for the next action a_{t+1} selection

The objective is for the RL agent to maximise the expected cumulative reward, or return,

$$G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k, \tag{3.1}$$

where r_k is the reward at time step k and $\gamma \in [0, 1]$ is the discount factor that determines the trade-off between immediate and future rewards. The expected return G_t allows the agent to account for long-term consequences of its actions, not just the immediate reward [7, 66].

In simple Markov environments, such as the grid world shown in Figure 3.2, it can

easily be seen that the optimal path to the goal of the golden star can be determined by observing the current position of the agent in the grid. Hence, it is unnecessary to know the events that led up to the current state of the agent. In this grid world example, we can also see that the agent has the possibility to only exist within one of the blocks; therefore, the agents state space can be described by an index in the range of 1 to 9. The agent's action space consists of the possible actions in the environment, which are moving up $\uparrow$, down $\downarrow$, left $\leftarrow$, or right $\rightarrow$.

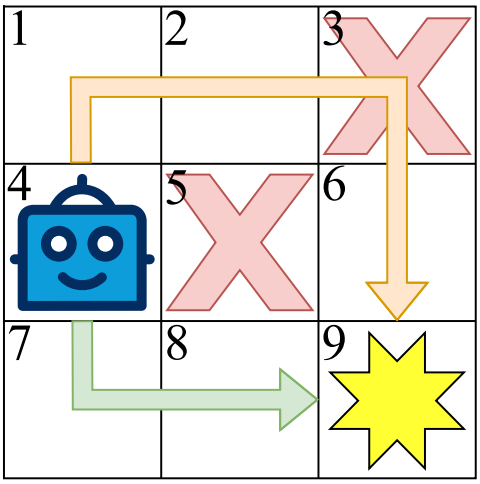


Figure 3.2: Grid world environment where the goal is to reach the gold star. The red crosses indicate a danger state where the agent can lose reward. Each block in the grid world is a possible agent state and is described with an index 1 to 9. The arrows indicate two possible paths, path 1 (orange) and path 2 (green)

Given this, to evaluate the quality of a state, we define the value function $V_\pi(s)$, which represents the expected return starting from state s and following policy π

$$V_\pi(s_t) = \mathbb{E}_\pi[G_t \mid S_t = s_t]. \tag{3.2}$$

where S_t represents all possible states and s_t describes the current state. By doing this, the value function identifies states that are more beneficial to be in [7, 66]. We can define the reward for reaching the goal state as 5. Subsequently, if the agent transitions into a danger state, it gets a reward of -10, and if it transitions to an empty state, it gets 0 reward. Based on the reward values and constraints, we can determine the actual reward of the agent in the grid world. In the grid, states surrounding the goal will have a higher value than those far away, that is, state 8 will have a higher value than state 1.

Similarly, the action value function, or the Q-function $Q_\pi(s, a)$, represents the expected return from state s_t, taking action a_t, and subsequently following policy π

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi[G_t \mid S_t = s_t, A_t = a_t]. \tag{3.3}$$

where A_t is all possible action and a_t is the action taken in that state [7].

The Bellman equation provides a recursive relationship for these value functions. For the state-value function, it can be expressed as

$$V_\pi(s_t) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(s_{t+1}) \mid S_t = s_t]. \tag{3.4}$$

This equation shows that the value of the current state is the immediate reward plus the discounted value of the next state. For the action-value function, the Bellman equation is similarly defined as

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi[R_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) \mid S_t = s_t, A_t = a_t]. \tag{3.5}$$

To illustrate this, we consider two possible paths to the goal state. Path 1 (indicated in orange in Figure 3.2) can be described as a sequence of states followed by the actions taken to reach the subsequent states. For example, $4 \uparrow$ indicates that the agent is in state 4 and takes the action to move up. The full path is described as

$$4\uparrow \quad 1\rightarrow \quad 2\rightarrow \quad 3\downarrow \quad 6\downarrow \quad 9$$

The return for this path, when $\gamma = 0.99$, can be calculated following Equation 3.1 as

$$G_t = 0 + \gamma(0) + \gamma^2(-10) + \gamma^3(0) + \gamma^4(5) = -5$$

Next path 2 is considered

$$4\downarrow \quad 7\rightarrow \quad 8\rightarrow \quad 9$$

The return here would be

$$G_t = 0 + \gamma(0) + \gamma^2(5) = 4.9$$

We can see that when in the state 4, it is much more beneficial to go down than up, as it will ultimately lead to more reward. This is the exact thing the Q-function aims to predict. The Q-function would predict that when in state 4, going down in more optimal than going up, therefore,

$$Q(4, \downarrow) > Q(4, \uparrow)$$

This is how Q-value estimates of the expected return based on the state and action can guide the agent to select an action that lead to more reward. Understanding Bellman equations is crucial as they serve as the basis for many reinforcement learning algorithms. By defining how future rewards are computed based on current actions and states, the Bellman equation enables the development of policies that optimise decision-making processes over time. This recursive nature not only simplifies the computation of value functions but also establishes a framework for learning optimal strategies in complex environments [7].

Although the simple and discrete nature of the grid-world problem effectively illustrates how the Q-function can be used to determine the most rewarding action given the current state, real-world applications of reinforcement learning often involve continuous state and action spaces. In these scenarios, traditional Q-learning approaches become impractical due to the vast number of possible states and actions, necessitating the development of more complex methods. Deep reinforcement learning (DRL) addresses this challenge by using deep neural networks (DNNs) as function approximators, enabling the agent to predict Q-values in a continuous domain [27, 66].

3.2 Deep neural networks

Deep neural networks (DNNs) form a subset of machine learning techniques that focus on using multilayer networks to map complex relationships between inputs and outputs [7]. These algorithms have recently demonstrated their ability to perform tasks in many domains from image processing to large language models. The advantage of DNNs is their ability to approximate non-linear functions even when there is noise present in the data, which makes them an invaluable tool when using noisy real-world data [27].

These networks consist of an input layer, hidden middle layers, and an output layer. The layers consist of artificial neurons. Each neuron in a layer receives an input x_i from neurons in the previous layer, a weighted sum w_i is then applied to these inputs, and a bias term b is added before getting the result

$$\hat{y} = \sum_{i=1}^{n} w_i x_i + b, \tag{3.6}$$

This output $\hat{y}$ is then passed through an activation function which allows the network to model complex, non-linear relationships between the input data and the target output [67]. These layers are fully connected, meaning that each node in one layer connects to all nodes in the next layer, making them dense layers. The hidden layers introduce non-linearity into the network, which allows DNNs to learn complex functions and representations of the data. A model of a layer and the activation function is shown in Figure 3.3.

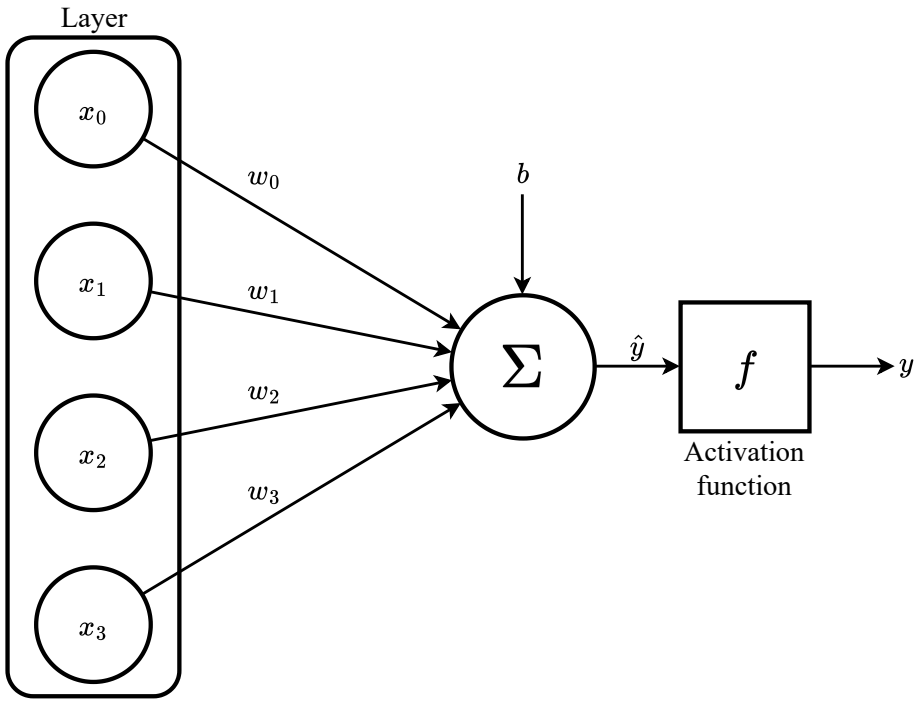


Figure 3.3: Basic model of a neuron showing the inputs x and their corresponding weights w summed with the bias b and passed to the activation function. The output y will be fed into each neuron in the next layers

In order for these DNNs to learn the mapping between the inputs and outputs, the neurons are updated after every forward propagation of the initial input through the network. The weight update is performed using a technique called backpropagation [67]. To do this, the loss function quantifies how well the prediction of the network matches the target value. This is called the loss L. The algorithm then calculates how much each weight contributed to this loss. This begins at the output layer and moves back through the network to the input layer. For a neuron in the output layer, the derivative of the loss with respect to the output is first calculated. Then, this derivative is propagated backwards through each layer, allowing the algorithm to compute the derivative of the loss with respect to the weights for each neuron as

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w}. \tag{3.7}$$

Once the gradients are computed, the weights are updated using gradient descent

$$w = w - \alpha \cdot \frac{\partial L}{\partial w} \tag{3.8}$$

where α is the learning rate which dictates how much the loss derivative contributes to the change in the weight. This iterative weight update of all neurons is what allows these networks to learn the complex nonlinear relationship between the input and output

variables.

This structure of DNNs is an integral part of many learning algorithms that form the backbone of Deep Reinforcement Learning (DRL), where they are utilised to approximate complex functions, enabling agents to effectively learn optimal behaviours through interactions with their environment. For racing applications, DNNs are particularly valuable in handling continuous action spaces and complex state representations.

3.2.1 Deep Q-networks

Deep Q-Networks (DQNs) are a class of deep reinforcement learning (DRL) algorithms that use deep neural networks to approximate the Q-function (Equation 3.3). In DQNs, the agent learns by minimising the temporal difference (TD) error, which is the difference between the predicted Q-values and the target Q-values derived from the Bellman equation. The TD error is calculated as the difference between the current estimate of the Q-value $Q(s_t, a_t)$ and the sum of the observed reward plus the maximum future Q-value,

$$TD_{error} = r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t), \tag{3.9}$$

where r_t is the reward received after taking action a_t in state s_t, γ is the discount factor, and $\max_a Q(s_{t+1}, a)$ is the estimated maximum future Q-value for the next state s_{t+1}. This error quantifies the gap between the current Q-value estimate $Q(s_t, a_t)$ and what the agent has just learnt from its interaction with the environment [67]. To minimise this error, the Q-function is updated iteratively using the update rule,

$$Q(s_t, a_t)_{\text{new}} \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right], \tag{3.10}$$

where α is the learning rate. This equation adjusts the Q-value estimate for the current state-action pair based on the TD error. Over time, by repeatedly applying this update rule, the Q-values become more accurate, and the agent's policy converges toward the optimal policy that maximises cumulative future rewards. The process of minimising the TD error is fundamental in guiding the agent's learning, ensuring that the Q-function reflects the cumulative expected reward and improves the agent's decision-making capabilities. Using the approximation ability of DNNs, DRL can handle high-dimensional state spaces and complex environments and can guide the agent to select the most optimal action given the current state. [7, 66, 68].

3.2.2 Policy gradient methods

While DQN methods focus on learning value functions to derive optimal policies, policy gradient methods take a different approach by directly learning the policy π that maps

states to actions. Instead of first computing value functions and then selecting actions based on these values, policy gradient methods directly optimise the policy parameters to maximise expected rewards. This direct approach makes them particularly suitable for continuous action spaces, such as those found in autonomous racing, where precise controls are required [27, 66]. In policy gradient methods, the policy π is represented by a neural network with parameters θ that directly outputs control actions. The fundamental objective in policy gradient methods is to maximise the expected cumulative reward,

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} r(s_t, a_t)] \tag{3.11}$$

where τ represents a trajectory (sequence of states and actions) sampled according to policy π_θ. The key theoretical foundation of these methods is captured in the policy gradient theorem, which provides an expression for the gradient of the expected return $J(\theta)$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t] \tag{3.12}$$

This gradient indicates how the policy parameters θ should be adjusted to increase the expected rewards. For each trajectory, $\nabla_\theta \log \pi_\theta(a_t \mid s_t)$ indicates how responsive the likelihood of selecting action a_t in state s_t is to minor changes in the policy parameters θ, effectively showing the influence of these parameters on action choices within the policy [7]. However, this term alone does not determine whether the action a_t is good or bad. This is where G_t, the total expected future reward, becomes important. G_t serves as a score for the action a_t based on subsequent rewards. If taking action a_t in state s_t leads to high future rewards, then G_t will be large, encouraging the policy to select similar actions in the future. In contrast, if G_t is small, indicating that the action resulted in poor outcomes, the update will work to reduce the probability of selecting that action in similar states [7, 27]. Thus, G_t guides the strength and direction of the policy update, enabling the agent to learn which actions are beneficial. The policy parameters are adjusted accordingly, increasing the likelihood of actions that maximise rewards. This process is captured in the policy update rule, where the parameters are updated through gradient ascent,

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta J(\theta_t), \tag{3.13}$$

where α represents the learning rate [7]. Policy gradient methods offer several compelling advantages in continuous action spaces. They can directly output continuous control actions, naturally learning smooth control policies that are essential for stable performance.

However, policy gradient methods are not without limitations. They often suffer from high variance in gradient estimates, which can lead to unstable training. They also tend to be sample-inefficient, requiring many interactions with the environment to learn

effective policies. Furthermore, these methods can struggle with credit assignment, which is determining which actions in a sequence were truly responsible for obtaining rewards. These limitations make policy gradient methods suboptimal as a standalone solution for complex racing problems [7, 27].

The limitations inherent in both DQN and policy gradient approaches motivate the use of actor-critic methods which combine the strengths of both methods. Actor-critic methods merge direct policy optimisation of policy gradient methods with the value function learning of DQN, while introducing additional techniques to address their respective limitations. This combination proves particularly effective for demanding continuous environments [7, 66].

3.3 The twin delayed deep deterministic policy gradient algorithm (TD3)

Having examined both value-based methods and policy gradient approaches, the twin delayed deep deterministic policy gradient (TD3) algorithm is introduced, which combines and improves upon these concepts. TD3 is an actor-critic algorithm that builds on its predecessor, the deep deterministic policy gradient (DDPG) algorithm, one of the first algorithms to merge value-based and policy-based learning for continuous control tasks. While DDPG implements an actor-critic architecture where the actor (policy network) directly maps states to actions and the critic (value network) evaluates these actions using Q-learning principles, it often suffers from training instability and sensitivity to hyperparameter tuning. A common issue is the tendency towards overestimation bias in the critic network, where Q-values are overestimated, leading to suboptimal policy learning [66].

TD3 addresses these limitations through several improvements that make TD3 particularly suitable for complex continuous control tasks, where stable learning and precise action selection are important [69, 70]. The algorithm uses six neural networks, two for the actor (model ϕ and target ϕ') and four for the critics (two models θ_1, θ_2 and two targets θ_1', θ_2'). The model networks, also known as primary networks, are used to select actions and calculate Q-values during each step of training. The target networks, in contrast, are periodically updated copies of the model networks. They provide a stable reference for updating the Q-values of the model network, helping to reduce volatility and improve learning stability. In this way, the target networks serve as a stable baseline for comparing and adjusting the model network's values. The actor networks implement a deterministic policy π, mapping states directly to actions, while the critic networks implement action-value functions Q, evaluating state-action pairs. This dual-critic architecture is important for the improved stability of TD3 [70].

The learning process begins by collecting experience tuples (s_t, a_t, r_t, s_{t+1}) in a replay buffer. During training, the algorithm selects actions using the current policy with added exploration noise,

$$a_t = \pi_\phi(s_t) + \epsilon, \tag{3.14}$$

where ϵ represents Gaussian noise with standard deviation σ,

$$\epsilon \sim \mathcal{N}(0, \sigma). \tag{3.15}$$

The target Q-value, which is used to predict the total return based on the information in the tuple, is computed using the minimum of both target critic Q-value estimates,

$$Q_{\mathrm{t}} = r_t + \gamma \min_{i=1,2}(Q'_{\theta'_i}(s_{t+1}, a_{t+1})), \tag{3.16}$$

where the next action a_{t+1} is found by giving the next state s_{t+1} to the target actor $\pi_{\phi'}(s_{t+1})$,

$$a_{t+1} = \pi_{\phi'}(s_{t+1}) + \tilde{\epsilon}, \tag{3.17}$$

where $\tilde{\epsilon}$ is clipped Gaussian noise with standard deviation $\tilde{\sigma}$

$$\tilde{\epsilon} \sim \mathrm{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c). \tag{3.18}$$

The critic networks are updated by minimising the mean squared error between the more accurate target Q-value, Q_t, and the models estimated Q-value,

$$\theta_i \leftarrow argmin_{\theta_i}\Big(2^{-1}\sum_{i=1}^{2}(Q_t - Q_{\theta_i}(s_t, a_t))^2\Big). \tag{3.19}$$

Here, the model attempts to predict the future return using only the current state-action pair (s_t, a_t). The target Q-value, Q_t, incorporates both the immediate reward r_t and information from the next state-action pair, (s_{t+1}, a_{t+1}). This access to future information makes the target Q-value a more accurate estimate of the true expected return.

The mean squared error between the target and the model's Q-values is used to update the model critic networks via backpropagation and gradient descent (as shown in Equation 3.8). The goal is for the model critic networks to learn how to predict the total return based on the current state and action, without needing the additional future information available to the target network. Over time, this enables the model critics to become good estimators of the future return based solely on the current state-action pair.

This role of the critic is crucial, as it serves to judge the quality of the actions chosen by the actor. If the critic believes that a poor action has been selected, it will assign a lower predicted return; conversely, if the action is deemed optimal, the predicted return will be higher [7, 27, 66].

The model actor network is then periodically updated by performing gradient ascent based on the first model critic predicted return,

$$\nabla_\phi J(\phi) = \frac{1}{N}\sum_{i=1}^{N} \nabla_a Q_{\theta_1}(s_i, a)\Big|_{a=\pi_\phi(s_i)} \nabla_\phi \pi_\phi(s_i). \tag{3.20}$$

The objective here is to improve the policy by maximising the expected return. The gradient of the Q-value, $\nabla_a Q_{\theta_1}(s, a)$, represents the critic's assessment of the action a chosen by the current actor policy $\pi_\phi(s)$ for the state s. This judgement guides the actor's learning process [69, 70].

Simultaneously, the gradient of the actor's policy, $\nabla_\phi \pi_\phi(s)$, is adjusted to improve the actions it chooses in the future. Since the policy $\pi_\phi(s)$ maps states to actions, this gradient is used to fine-tune the model actor's parameters ϕ, ensuring that the actions chosen by the actor lead to higher Q-values, i.e., better expected future rewards.

By periodically updating the actor, critic networks have enough time to converge to accurate value estimates before their feedback is used to improve the actor's policy. This ensures a more stable learning process, where the actor can rely on well-trained critics to guide its policy updates effectively [69, 70].

TD3 is an ideal algorithm for this application, as the use of dual critic networks directly addresses the overestimation bias present in DDPG. By taking the minimum of two Q-value estimates in Equation 3.16, TD3 implements a form of double Q-learning that provides more conservative value estimates.

To prevent the policy from exploiting Q-function errors in individual states, TD3 uses target policy smoothing by adding noise to the target actions seen in Equation 3.17. This smoothing regularisation effectively trains the policy on a mini-distribution of target values, making it more robust to noise and helping prevent overfitting to specific state-action pairs [27, 66]. The delayed policy updates allow critics to become more accurate before they are used to improve the policy, reducing the likelihood of learning from inaccurate value estimates. The target networks are updated using Polyak averaging

$$\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i' \tag{3.21}$$

and,

$$\phi' \leftarrow \tau\phi + (1-\tau)\phi', \tag{3.22}$$

where $\tau \in [0, 1]$ is an update weight influencing the rate at which the targets get updated. This allows for smooth target updated by gradually updating these parameters rather than abruptly changing them. The complete TD3 implementation can be seen in Algorithm 3.1 which outlines the sequence in which the processed discuses are executed.

TD3's architecture makes it particularly well-suited for autonomous racing tasks. The continuous action space of racing benefits from the deterministic policy approach, while

Algorithm 3.1: Twin Delayed Deep Deterministic Policy Gradient (TD3) adapted from Dankwa and Zheng [69]

Initialise random critic networks $Q_{\theta_1}, Q_{\theta_2}$ and actor network π_ϕ
Initialise target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Define target update frequency F
Initialise random replay buffer $\mathcal{D}$
for each timestep **do**
    Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$
    Observe next state s_{t+1} and reward r
    Store transition (s, a, r, s_{t+1}) in $\mathcal{D}$
    Sample mini-batch of transitions from $\mathcal{D}$
    Add noise to target action: $a_{t+1} = \pi_{\phi'}(s_{t+1}) + \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
    Compute target Q-value: $Q_t = r + \gamma \min_{i=1,2} Q_{\theta'_i}(s_{t+1}, a_{t+1})$
    Update critics: $\theta_i \leftarrow argmin_{\theta_i}\left(2^{-1}\sum_{i=1}^{2}(Q_t - Q_{\theta_i}(s_t, a_t))^2\right)$
    if timestep % F **then**
        Update actor:$\nabla_\phi J(\phi) = \frac{1}{N}\sum_{i=1}^{N} \nabla_a Q_{\theta_1}(s_i, a)\Big|_{a=\pi_\phi(s_i)} \nabla_\phi \pi_\phi(s_i).,$
        Update target networks:
        $\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$
        $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$
    end if
end for

dual critics and delayed updates provide the stability needed for learning complex racing behaviours. The target policy smoothing helps the agent generalise across states, as this noise encourages the development of a robust policy, which is crucial for handling varying tracks and racing scenarios. Furthermore, the use of experience replay allows the agent to learn from various racing situations, preventing overfitting to specific track sections or racing scenarios. This combination of features enables TD3 to learn robust racing policies that can be generalised to unseen track configurations while maintaining stable training dynamics [69, 70].

When using a DRL algorithm like TD3 for a complex task such as racing, considerations have to be made when discussing the agent state space and action space. Formally, the agent state space is all possible states in which the agent can exist in the environment. When operating in a discrete environment such as the grid in Section 3.1 the agent's state can easily be described by the index of all possible states $s \in S$. This is not the same when operating in a complex continuous environment, as there is no way to know all the possible states. To account for this, the agent's state s can be described using a state vector $\mathbf{s}$ that contains variables that describe its condition and the environment around it. This vector is then used as the actionable information that the agent uses to make a decision. Similarly, the agents' actions are no longer discrete single choices as in the grid; therefore, the actions selected by the agents are in the form of an action vector $\mathbf{a}$ that contains specific actions available in the agent action space A.

Chapter 4

Autonomous racing: Problem conceptualisation and simulation

Racing is a complex task that requires precise metrics to evaluate and benchmark performance. In this chapter, we model the racing problem by examining the scenarios presented in both seen and unseen track environments. We introduce the performance metrics that will be used to evaluate the effectiveness of autonomous algorithms in these scenarios, providing a basis for comparison and analysis. In addition, the vehicle model and its interaction with the simulated environment are described. Finally, the controller used to translate the algorithm outputs into the model input is discussed.

4.1 Conceptualising the autonomous racing problem

Racing is traditionally known as a competitive sport that relies on a driver's ability to make split-second decisions, navigate the complexities of different track layouts, and optimise speed while maintaining control of the vehicle. In contrast, autonomous racing involves the design of algorithms that replace human drivers with software capable of making these decisions. The task of autonomous racing introduces challenges that differ from traditional racing; however, the goal is the same. The primary objective is to navigate a vehicle around the track as quickly and safely as possible.

In single-vehicle autonomous racing, the objective is to move the vehicle from a standstill to the completion of a lap. This is achieved by providing the vehicle with control commands that regulate its behaviour. Typically, these commands consist of vehicle speed v and steering angle δ. These two control inputs are widely used because they capture the essential aspects of vehicle control [35, 40, 71, 72]. Furthermore, they mirror how human drivers control a car by adjusting the speed through throttle input and steering through the steering wheel.

With the goal of autonomous racing defined, the next element in the racing problem is the track itself. A track provides the environment within which the vehicle operates. In this context, a track is represented by an image that captures the bird's eye view of its

outline, along with a 2D matrix of centerline coordinates

$$\mathbf{C} = \begin{pmatrix} x_0 & y_0 \\ x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \end{pmatrix}$$

In this matrix, each row represents a position along the track's centerline, beginning at (x_0, y_0) and extending for n points before looping back near the starting position.

When racing on a seen track, the outline, centerline, and starting point are known in advance, meaning that any of this information can be used to help navigate the vehicle around the track during testing. In contrast, when racing on an unseen track, none of this information is known at the time of testing. The vehicle is placed at the start point of the unseen track and does not have any information about the complete layout of the track.

When the vehicle is placed on the track, it can be described by its pose. The pose of the vehicle consists of state variables that describe position, x, y, speed v, steering angle δ, and heading Ψ. This information is sufficient to describe its location on the track and its current motion. Furthermore, the environment around the vehicle is described by sensor measurements. The sensor chosen for this application is a LiDAR as it is a robust and accurate vehicle sensor, as well as the main sensor on an F1TENTH vehicle [3]. The LiDAR describes the environment using a scan vector $\mathbf{d}$ that contains distance measurements from each of the individual LiDAR beams in the scan

$$\mathbf{d} = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 & \dots & d_n \end{bmatrix}, \tag{4.1}$$

where n represents the number of measurements in the scan.

Figure 4.1 shows how the LiDAR maps the environment by measuring the distance of each of the beams.

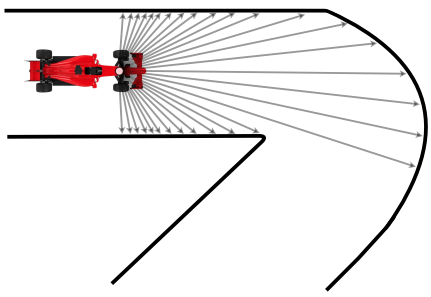


Figure 4.1: The vehicle on the track showing the LiDAR array $\mathbf{d}$ indicated by the grey arrows

4.2 Racing metrics

Now that the vehicle can be described in the environment with a pose and a LiDAR scan, metrics to describe racing performance have to be established. An important metric is the completion status of a lap, which is determined by whether the vehicle crosses the finish line without colliding with the track boundary. The detection of a collision is modelled by checking all of the elements of the LiDAR array and ensuring they are not measuring a distance shorter than the safety distance d_s,

$$d_i < d_s, \text{ where, } d_i \in \mathbf{d} \tag{4.2}$$

This is illustrated in Figure 4.2, where the safety distance is defined relative to the position of the LiDAR above the front wheels. In this racing problem, the vehicle is intended to move forward continuously, making it sufficient to focus collision checks on the front. Monitoring for potential rear collisions is unnecessary, as our vehicle is not permitted to reverse during the race. This collision detection model can also be used in a simulated and a real environment, as it does not rely on any information that is only accessible in simulation, such as reading states from the track boundary itself.

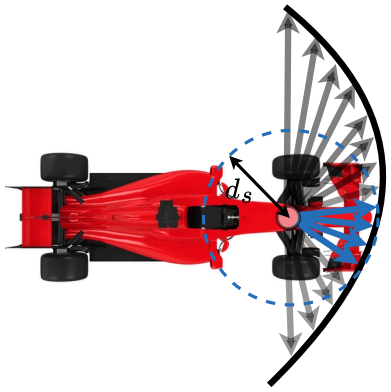


Figure 4.2: The safety circle with radius d_s shows the LiDAR beams (blue) that detect a collision between the vehicle and the boundary

Another important metric that gives more insight into where a collision occurs is the progress along the track. Progress is tracked using the vehicle's current position in relation to the beginning of the track. This is shown in Figure 4.3 where the vehicle's current progress is shown in red. The centerline point nearest to the vehicle is used to calculate the distance from the beginning of the track to the vehicle's current point. Progress is the proportion of the current length travelled l_t over the length of the entire track l_{total}

$$p_t = \frac{l_t}{l_{total}}. \tag{4.3}$$

As the whole length of the track is required to calculate progress, it is a metric that can only be used on seen tracks. This metric allows us to track the safety and consistency of the racing algorithms, but does not give much indication of the racing performance of the algorithms.

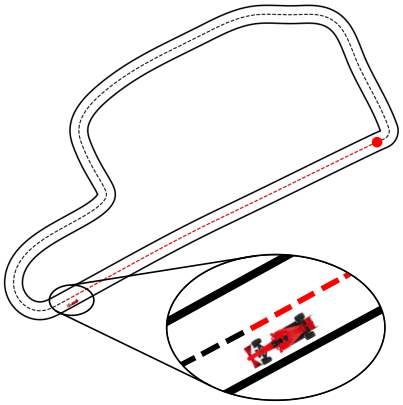


Figure 4.3: The current progress made by the vehicle from the start of the track as indicated by the red dot. The zoomed portion shows the closest centre point to the vehicle

Lap time L_T is a widely accepted metric for evaluating racing performance, integrating other performance factors such as trajectory, speed profile, steering efficiency, and slip angles. A faster lap time is generally indicative of better vehicle behaviour and better performance in these variables. These three metrics, collisions/completion rate, progress, and lap time, are commonly used to set benchmarks for racing performance [33, 35, 71, 72].

Although real-world training is ideal due to its direct applicability, practical constraints such as increased training time, potential component damage, and track availability deter researchers from doing this. Instead, simulated environments provide a safer and more efficient alternative. The simulated environment serves as an effective platform for developing and testing autonomous racing algorithms, providing a realistic, safe, and efficient space for researchers to test and experiment with new racing algorithms. The simulator effectively mimics vehicle motion using a model that accurately represents real vehicle dynamics, allowing adequate testing of algorithms within an environment closely resembling their intended use.

4.3 The F1TENTH simulator

The safety and computational advantages of developing algorithms in simulation make it an ideal environment for testing and refinement. For this purpose, the F1TENTH simulator (introduced in Chapter 1) is used, which incorporates a vehicle model designed to mimic real F1TENTH vehicles [3]. This allows algorithms to be safely developed in simulation and then tested on a real vehicle.

4.3.1 Simulated vehicle model

The F1TENTH vehicle used is modelled using the standard bicycle model to describe the dynamics of the vehicle [15]. This simplified model uses only two wheels to represent the vehicle. This simplification has a limitation in that it does not consider the roll dynamics; however, the vehicle's low profile and the speed at which it operates generally negate this limitation's effect. Figure 4.4 shows the model and the accompanying parameters used as state variables.

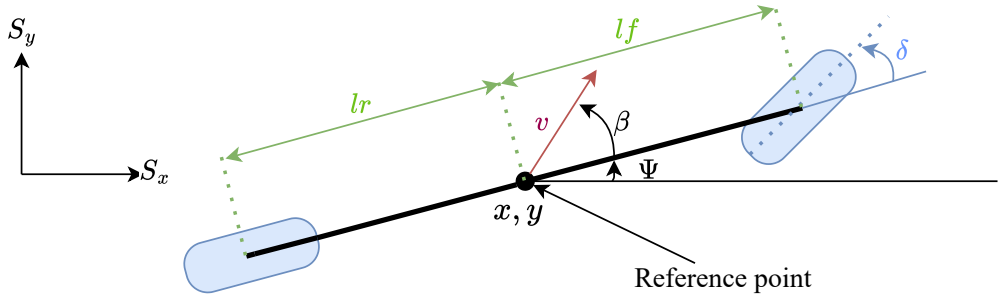


Figure 4.4: The standard bicycle model [15] with the centre of mass as the reference point. The diagram shows the position x, y, speed v, steering angle δ, slip angle β and the heading Ψ as well as how they are defined relative to a fixed plane S_x, S_y

The vehicle state-space vector

$$\mathbf{x} = \begin{bmatrix} x & y & \delta & v & \Psi & \dot{\Psi} & \beta \end{bmatrix}^T \tag{4.4}$$

serves as a representation of the vehicle's current state under the single-track model. The vehicle model is only accurate when operating within the defined limits of its constraints. However, the common speeds of F1TENTH vehicles with autonomous racing algorithms range between 4 and 8 m/s [71] and the steering angles between -0.4 and 0.4 rad [73]. Consequently, the more specific constraint equations applied when the speed and steering angle are at the extremes of the constraint bounds do not apply in this use case. The constraints are shown in Table 4.1 which show that the model constraints are much larger than the ranges expected in this project.

However, some constraints still apply to the input vector. In this model, the primary control inputs are the steering rate ω and acceleration a, which determine the rate of change in the steering angle δ and speed v respectively

$$\mathbf{u} = \begin{bmatrix} \omega & a \end{bmatrix}^\top \tag{4.5}$$

Table 4.1: Hardware imposed constraints of steering and speed

Constraint Type	Parameter	Symbol	Value
Steering Constraints	Minimum steering angle	$\underline{\delta}$	-0.8 [rad]
	Maximum steering angle	$\overline{\delta}$	0.8 [rad]
	Minimum steering speed	$\underline{\omega}$	-0.4 [rad/s]
	Maximum steering speed	$\overline{\omega}$	0.4 [rad/s]
Longitudinal Constraints	Minimum speed	$\underline{v}$	-13.6 [m/s]
	Maximum speed	$\overline{v}$	50.8 [m/s]
	Switching speed	v_s	7.319 [m/s]
	Maximum acceleration	$\bar{a}$	11.5 [m/s^2]

Following the notion used by Althoff and Würsching [15], where the maximum limit is defined by using an overline bar $^-$ and the lower limit is defined by an underline bar _. The input $\mathbf{u}$ is subjected to the constrains

$$|\omega| \in [0, \overline{\omega}], \quad |a| \in [0, \overline{a}(v)], \tag{4.6}$$

where a piecewise function defines $\bar{a}(v)$ as

$$\overline{a}(v) = \begin{cases} \bar{a} \cdot \frac{v_s}{v} & \text{for } v > v_s, \\ \bar{a} & \text{otherwise.} \end{cases} \tag{4.7}$$

In this function, v_s denotes the speed above which tyre friction is no longer the limiting factor, but rather the acceleration is limited by the power of the engine [15].

The state matrix for the model

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\delta} \\ \dot{v} \\ \dot{\Psi} \\ \ddot{\Psi} \\ \dot{\beta} \end{bmatrix} = \begin{bmatrix} v\cos(\Psi + \beta) \\ v\sin(\Psi + \beta) \\ f_\omega(\omega) \\ f_a(a) \\ \dot{\Psi} \\ f_{\ddot{\Psi}}(\delta, \beta, \dot{\Psi}, v) \\ f_{\dot{\beta}}(\delta, \beta, \dot{\Psi}, v) \end{bmatrix} \tag{4.8}$$

defines the relationship between the system's current state and its future state, effectively capturing the internal dynamics of the system. The rate of change of steering $f_\omega(\omega)$ and acceleration $f_a(a)$ are constrained by a sequential piecewise function, which aims to keep the variables within its operational limits

$$f_\omega(\omega) = \begin{cases} \bar{\omega} & \text{if } |\omega| \geq \bar{\omega} \\ \omega & \text{otherwise} \end{cases} \tag{4.9}$$

$$f_a(a) = \begin{cases} \overline{a}(v) & \text{if } |a| \geq \overline{a}(v), \\ a & \text{otherwise} \end{cases} \tag{4.10}$$

The functions for $\ddot{\Psi}$ and $\dot{\delta}$ are functions of the state variables and use the constants given in Table 4.2 that model the rate of change of heading and rate of change of slip angle,

$$\begin{aligned} f_{\ddot{\Psi}}(\delta, \beta, \dot{\Psi}, v) = \frac{\mu m}{I_z(l_r + l_f)} \Bigg[& l_f C_{S,f}(g l_r - a h_{cg})\delta \\ & + \Big(l_r C_{S,r}(g l_f + a h_{cg}) - l_f C_{S,f}(g l_r - a h_{cg}) \Big)\beta \\ & - \Big(l_r^2 C_{S,r}(g l_f + a h_{cg}) + l_f^2 C_{S,f}(g l_r - a h_{cg}) \Big)\frac{\dot{\Psi}}{v} \Bigg], \end{aligned} \tag{4.11}$$

$$\begin{aligned} f_{\dot{\beta}}(\delta, \beta, \dot{\Psi}, v) = \frac{\mu}{v(l_r + l_f)} \Bigg[& C_{S,f}(g l_r - a h_{cg})\delta \\ & - \Big(C_{S,r}(g l_f + a h_{cg}) - C_{S,f}(g l_f - a h_{cg}) \Big)\beta \\ & + \Big(C_{S,r}(g l_f + a h_{cg}) l_r - C_{S,f}(g l_f - a h_{cg}) l_f \Big)\frac{\dot{\Psi}}{v} \Bigg] - \dot{\Psi}. \end{aligned} \tag{4.12}$$

These vehicle parameters play an important role in defining the vehicle's behaviour during simulations, particularly in controlling aspects like cornering stiffness and inertia.

For small velocities, the single-track model becomes ill-defined, as certain terms in the equations involve division by velocity, leading to singularities as the velocity approaches zero. Therefore, for these low velocities, we switch to the kinematic single-track model with the centre of mass being the reference points as in the normal single-track model. The kinematic single-track model does not consider vehicle slip; therefore, aspects related to oversteer and understeer are not considered. This is a reasonable assumption as these effects are not dominant when the vehicle is not driving close to its physical limits. This assumption causes in the following changes to the state-space model for $|v| < 0.5$ where the last two state-space terms are altered to

$$f_{\ddot{\Psi}}(v, a, \beta, \delta, \dot{\Psi}, \omega) = \frac{1}{l_{wb}} \left[f_a(v, a)\cos(\beta)\tan(\delta) - v\sin(\beta)\tan(\delta)\dot{\Psi} + \frac{v\cos(\beta)}{\cos^2(\delta)} f_\omega(\delta, \omega) \right] \tag{4.13}$$

$$f_{\dot{\beta}}(\delta, \omega) = \frac{1}{1 + \left(\tan(\delta)\frac{l_r}{l_{wb}}\right)^2} \frac{l_r}{l_{wb}\cos^2(\delta)} f_\omega(\delta, \omega) \tag{4.14}$$

Table 4.2: Vehicle parameters for the single-track model obtained from the F1TENTH simulator

Name	Symbol	Unit	Value
Vehicle length	l	[m]	4.508
Vehicle width	w	[m]	1.610
Total vehicle mass	m	10^3 [kg]	1.093
Moment of inertia about z-axis	I_z	10^3 [kg m^2]	1.791
Distance from CG to front axle	l_f	[m]	1.156
Distance from CG to rear axle	l_r	[m]	1.422
CG height of total mass	h_{cg}	[m]	0.574
Cornering stiffness coefficient (front)	$C_{S,f}$	[1/rad]	20.89
Cornering stiffness coefficient (rear)	$C_{S,r}$	[1/rad]	20.89
Friction coefficient	μ	[-]	1.048
Gravity	g	[m/s^2]	9.81

4.3.2 Simulated input controllers

The racing problem requires the use of speed v and steering angle δ as primary control inputs. These control actions are translated into the corresponding vehicle state inputs $\mathbf{u}$ via a controller. A proportional controller is used to determine these inputs, specifically adjusting the vehicle's acceleration a and the steering rate $\dot{\delta}$ according to the desired speed v_d and steering angle δ_d. The proportional controller for the motors calculates the acceleration as

$$u_2 = a = k_p \cdot (v_d - v), \tag{4.15}$$

where,

$$k_p = 10 \cdot \frac{\overline{a}}{\overline{v}}, \tag{4.16}$$

which defines the difference between desired speed v_d and the current speed v. The rate of change in steering, which is the u_1 input to the state space equations, is defined by the maximum allowable steering rate $\overline{\omega}$, and the change in the direction of the steering angle,

$$u_1 = \omega = \frac{\Delta\delta}{|\Delta\delta|} \cdot \overline{\omega}, \tag{4.17}$$

where,

$$\Delta\delta = \delta_d - \delta. \tag{4.18}$$

These can then be used as inputs for the state space equations. This produces a rate of change for each of the state variables. Then Euler integration is performed to update the state variable.

$$\mathbf{x} = \mathbf{x} + \Delta t \cdot \dot{\mathbf{x}}. \tag{4.19}$$

The flow diagram in Figure 4.5 illustrates the process of action selection, where the autonomous racing algorithm selects an action, and the vehicle's state is updated through the control and state-space equations.

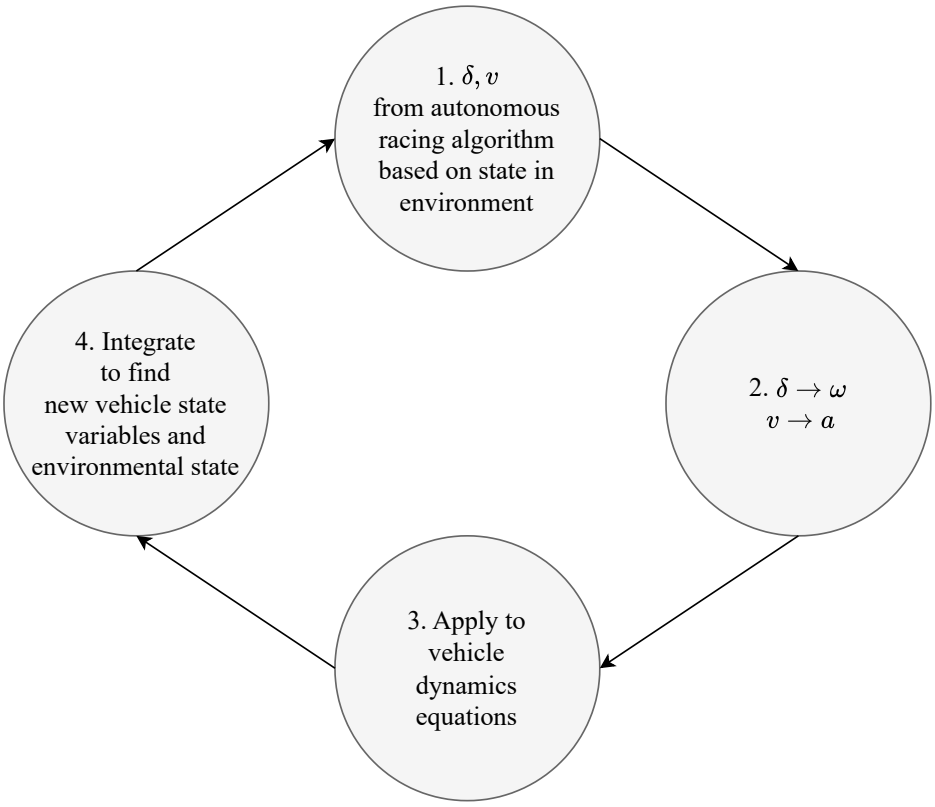


Figure 4.5: A flow diagram showing the processes from action selection to that action being applied to the vehicle model (processes 1-4). (1) It begins at the top block with an action selected by the autonomous racing algorithm. (2) Then the derivatives of these actions are found using the controllers (Equation 4.15, Equation 4.17) as they are inputs to the state equations. (3) The inputs are then applied to the state space equations. (4) Lastly, integration is done to update the vehicle state variables and environmental state. That completes the cycle as the agent then uses this state to select a new set of actions (1)

Understanding the racing problem and the goals associated with it are crucial to the development of an algorithm to achieve these goals. Having concrete metrics, such as collision status, progress, and lap time, to describe and compare the performance of these autonomous racing algorithms, is essential to track the success of these algorithms

and benchmark performance. A simulated environment, such as the one provided by F1TENTH, is a useful tool for collecting metrics to ensure that performance meets a certain standard before deploying these algorithms on real vehicles. Using a platform that allows for a direct comparison between a simulated and real environment further facilitates the progress of the development of these algorithms, as the algorithm's ability to perform in the real work is a key component to its success.

Chapter 5

Reinforcement learning formulation and optimisation for autonomous racing

In this chapter, the specific details of the twin delayed deep deterministic policy gradient (TD3) algorithm for racing are discussed. This includes aspects such as the composition of the agent's state vector, action vector, and reward function. These are the key components that determine the functionality of racing agents. The reward function weights and network hyperparameters are then tuned to find the best configuration for the agents.

5.1 Agent action vector

The agent is required to select commands to control the motion of the vehicle. It does this using the actor network π_ϕ to select actions from the action space, which consists of a set of all possible actions available to the agent. In the racing problem, the agent selects a steering angle δ and a speed v to allow it to navigate the track. The agent selects a value in the range of -1 and 1 for both the steering angle δ_{norm} and the speed v_{norm}. These specific choices form the action vector $\mathbf{a_t}$. In reinforcement learning, normalising the action and state vectors is a common practice [7]. It helps stabilise the training procedure, as inputs and outputs are within a consistent range. Additionally, having large values in either of these spaces can cause gradient updates to become unstable and biased. Lastly, having a normalised action vector allows the agent to better explore its action space and negates the possibility of the agent favouring actions with very large or small values [74]. The action vector is represented as follows:

$$\mathbf{a_t} = \begin{bmatrix} v_{t,norm} & \delta_{t,norm} \end{bmatrix}^\top \tag{5.1}$$

These actions are then scaled based on their maximum values,

$$\delta_t = \delta_{t,norm} \cdot \overline{\delta} \tag{5.2}$$

$$v_t = |v_{t,norm}| \cdot \overline{v} \tag{5.3}$$

These are the values the agent uses to control the vehicle. In order to optimise the actions to have the best racing performance, the agent has to have a good understanding of the current environment and vehicle state. To achieve this, information is presented to the agent in order to aid in the action selection process.

5.2 Agent state vector

The agent's state space consists of a collection of all possible states in which the agent could be in the environment. It uses the specific state in which it is in to select actions. Therefore, the information present in the agent's state space is directly linked to its ability to successfully race around tracks. As discussed in Section 3.3, there are clear distinctions between terms such as agent state space S_t, the agents current state s_t, and agent state vector $\mathbf{s_t}$ recalling that the state vector consists of variables used to describe the agent's current state. These variables can be in the form of an observation vector $\mathbf{o_t}$, or come from the vehicle state $\mathbf{x_t}$. The vehicle's state is described using the variables in its state equation Equation 4.4. These are used to describe the vehicle pose in the environment. An observation is a measurement of these vehicle states or the state of the environment. In practice, these observations are sensor measurements of these variables. The agent's state vector $\mathbf{s_t}$ contains information about the vehicle or environment that the agent uses to make decisions. If all of the vehicle state variables and all observations are included in the agents state vector,

$$\mathbf{s_t} = \begin{bmatrix} \mathbf{x_t} & \mathbf{o_t} \end{bmatrix}^\top \tag{5.4}$$

then it can be considered as Markov decision processes (MDP), where the states satisfy the Markov property (Section 3.1). However, if only a subset of the vehicle states or observations are included in the agent's state vector, then it is considered a partially observed Markov decision process (POMDP).

As the goal is to deploy this algorithm on the real vehicle, where the access to all of the vehicle state variables is limited, we will use a POMDP where a processed observation of the sensor measurement describing the vehicle's state and an observation of the environment is used.

$$\mathbf{s_t} = \begin{bmatrix} \mathbf{o_t} \end{bmatrix}^\top \tag{5.5}$$

As stated above, the agent's state vector must provide the agent with sufficient detail of the environment to make informed decisions. When assessing the information required for racing, it is clear that the position of the vehicle on the track is a crucial piece of information. In classic methods, this is often provided through a set of x, y coordinates along with an optimal trajectory composed of waypoints in the same format. As the goal

is for the agent to race on unseen tracks, where this information will not be accessible to the agent, alternative methods are required to attempt to recreate this information.

As identified in Section 4.1, the main sensor for gathering environmental information is a LiDAR sensor. Using the LiDAR sensor, we are able to gather data about the environments in the form of a LiDAR scan. This can be used to determine the current position of the vehicle; however, since the agent does not have other information about the global reference frame in which it and the track exist, this provides no value to the agent. Alternatively, given a complete LiDAR scan, the agent can infer information about its location based on its proximity to the track boundary, obstacles, and upcoming turns. Thus, enabling the agent to determine its position based on a local frame generated by the surroundings detected by the LiDAR. This local frame-of-reference eliminates the need for global position data, enhancing the agent's ability to generalise across diverse, unseen tracks. The spatial data from the LiDAR allow the agent to navigate safely while optimising for fast lap times, providing an essential input for both decision making and position control.

The LiDAR scan $\mathbf{d}$ is normalised $\mathbf{d}_{norm}$ (with respect to a maximum distance of 10 m) to ensure that the values in the state are between 0 and 1. The number of beams used in the scan is based on Ivanov et al. [35] findings which we used to establish that a scan of 28 beams

$$\mathbf{d}_{norm} = \begin{bmatrix} d_0 & d_1 & d_2 & \dots & d_{27} \end{bmatrix}^\top \tag{5.6}$$

with a 180° field of view (FOV) resulted in the best performance. This resolution is able to sufficiently describe some of the more complex track geometries causing more consistent behaviour.

Although the LiDAR provides valuable information about the vehicle's relative position, additional state variables would further aid the agent in making better action selections. Ideally, all relevant vehicle state variables would be included in the agent's state vector, as they are accessible in the simulator and provide a comprehensive overview of the vehicle's current motion. However, practical considerations must be made regarding the variables that can be measured on the real vehicle. Variables such as heading Ψ, rate of change in heading $\dot{\Psi}$, and slip angle β are not measurable with the sensors available on the vehicle and, therefore, cannot be included. On the other hand, speed v and steering angle δ can be estimated, as they are directly related to the hardware components of the vehicle.

These two state variables are also the most important when it comes to action selection, as they represent the actions selected by the agent. These variables offer the agent a more comprehensive understanding of the vehicle state than the LiDAR alone, potentially allowing the agent to make more accurate decisions and improve overall race performance. It is clear why it could be a very influential factor. The vehicle's current speed and steering angle should always be accounted for as it impacts the effect of the next selected action.

Large and rapid changes in speed and steering angle are undesirable as they are more likely to result in the agent operating at the limits of its physical constraints such as its tyre friction limit. When the agent has access to its current speed and steering angle, the effect of rapid and large changes in actions can be observed, and consequently, the agent can better learn to optimise its action selection. These values, $v_{m,raw}$ and $\delta_{m,raw}$, are measured with the sensors and are normalised before being concatenated to the state vector,

$$v_m = \frac{v_{m,raw}}{\overline{v}} \tag{5.7}$$

and,

$$\delta_m = \frac{\delta_{m,raw}}{\overline{\delta}}. \tag{5.8}$$

A common pitfall of end-end racing algorithms is a slaloming and jerking motion when racing which usually results in agents underperforming compared to classical methods [73]. This also decreases safety by increasing the chance of collisions as the margin of error is reduced because of its proximity to the track boundary. In an attempt to mitigate this problem, a centre term is added to the agent's state vector to be used as a reference when positioning the vehicle along the track. This method is derived from other methods that employ similar feature engineering, such as Evans et al. [36] using trajectory-aided learning (TAL), where a trajectory is given to the agent to encourage it to learn better positioning on the track, or Fuchs et al. [72] giving the agent upcoming track curvatures. This proposed centring method does not rely on having access to the track before the time and generating an optimal trajectory like TAL, but rather it calculates this centring term in real time. This ensures that this can be used on unseen tracks with diverse track geometries. This term is obtained using the LiDAR scan. This centring term c is acquired by averaging the first three and last three distances in the LiDAR array $\mathbf{d}_{norm}$ with n number of beams,

$$c = \frac{1}{3}\left(\sum_{i=1}^{3} d_i - \sum_{j=n-2}^{n} d_j\right). \tag{5.9}$$

Figure 5.1 identifies the red beams that are used to calculate c with different track geometries. Using the average of the three beams limits the effect of the track's boundary geometry and noise in the LiDAR scan. The addition of this centring term to the state vector creates what we refer to as centre-orientated TD3 (CO-TD3) agents.

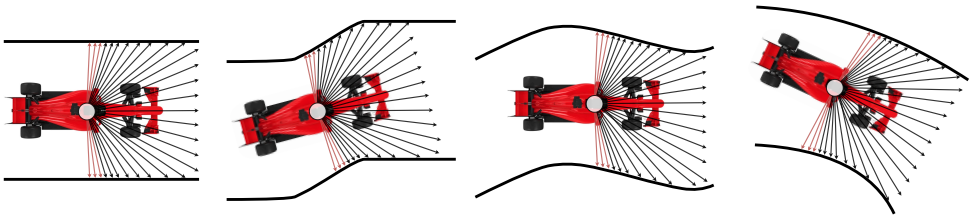


Figure 5.1: Red LiDAR beams used for centring calculations

This inclusion of the speed, steering angle, and reference term produces the agent's state vector,

$$\mathbf{s_t} = \begin{bmatrix} d_{t,0} & d_{t,1} & \dots & d_{t,n} & c_t & v_{t,m} & \delta_{t,m} \end{bmatrix}^\top, \tag{5.10}$$

which is the input to the actor $\pi_\phi(\mathbf{s_t})$ seen in Figure 5.2, which produces the next action.

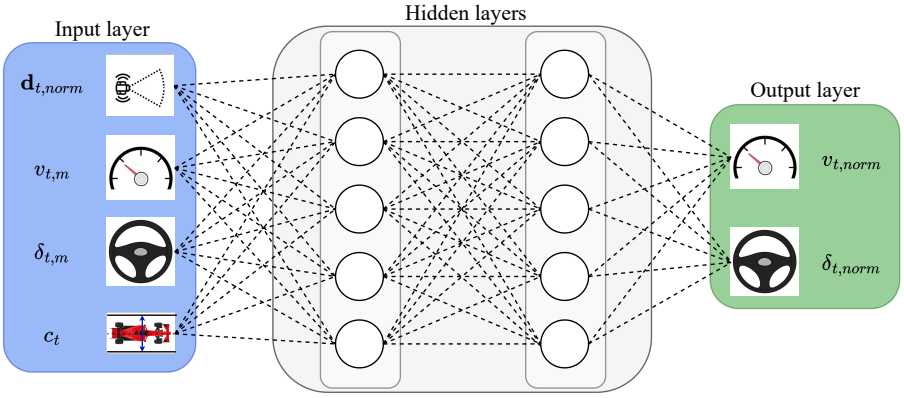


Figure 5.2: The agent structure shows the state vector terms in the input layer, the hidden network layers that map the inputs to the outputs, and the output layer shows the agent's action vector terms.

The combination of the actor's input and output forms the input for the critic networks $Q_{\theta_1}(\mathbf{s_t}, \mathbf{a_t}), Q_{\theta_2}(\mathbf{s_t}, \mathbf{a_t})$. This is used to produce an estimate of the expected return (Equation 3.16) based on the state and the action selected by the actor. The critic network, shown in Figure 5.3, is responsible for making the actor network better by using this predicted Q-value to update the parameters of the actor (Equation 3.20).

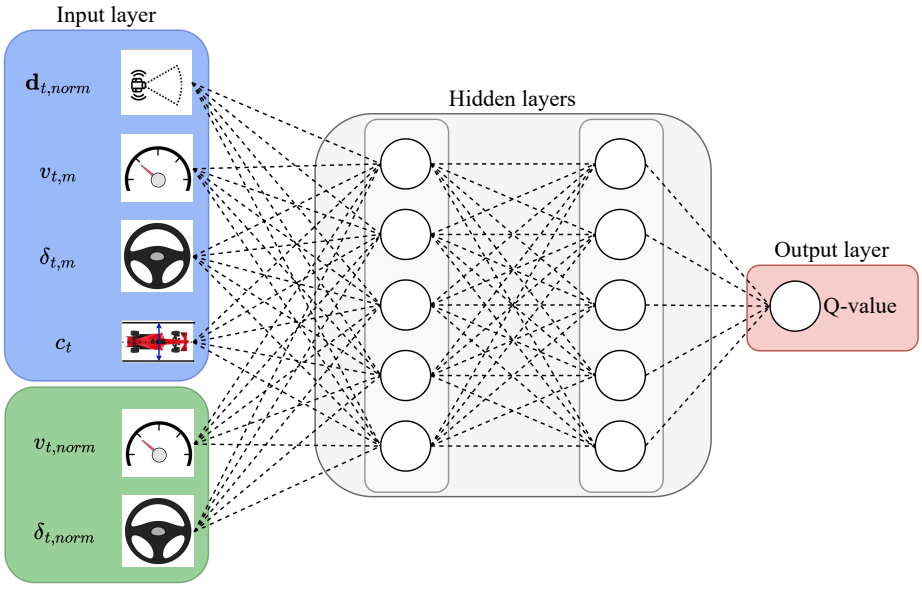


Figure 5.3: The structure of the critic network that produces a value to judge the performance of the actor network

Now that the inputs and outputs of the model and critic networks have been defined, they still required guidance to learn which actions produce the most expected cumulative reward.

5.3 Reward architecture

The reward function encourages the agent to perform in a certain way and is an integral part of the learning process. A well-structured reward function is necessary for the agent to achieve its main objective. The objective of our agent is to complete laps as fast as possible without crashing. Therefore, the reward function must encourage behaviours that enable it to achieve this goal by maximising its total cumulative reward.

The reward function used for our agents is comprised of different elements, each used to encourage certain racing behaviours.

$$r = r_p - r_c - |c| + r_f. \tag{5.11}$$

In this reward function, r_p represents the reward given based on progress p made per time step t. This progress is tracked using the vehicle's current position from the beginning of the track and the closest centerline point to the vehicle as in Equation 4.3. Therefore the reward term

$$r_p = (p_t - p_{t-1}) \cdot 10 \tag{5.12}$$

encourages the agent to make progress along the track.

To discourage the agent from crashing, a negative reward of weight w_1 is given if the vehicle crashes. A crash is modelled using the same methods as in Equation 4.2 which produces the reward function,

$$r_c = \mathbb{I}(d_i < d_s) \cdot w_1, \text{ where, } d_i \in \mathbf{d}. \tag{5.13}$$

The crash safety distance d_s is the smallest possible distance that encompasses vehicle which, for the position of the LiDAR on the F1TENTH vehicle, is 0.1 m.

To encourage safer progression and discourage slaloming, the agent is penalised for straying away from the centre of the track. The same methodology is used to quantify this penalty as Equation 5.9. Incorporating c in the reward function encourages the agent to drive more smoothly and eliminates slaloming; however, it decreases the agent's average speed. This smooth trajectory is beneficial, as it will result in a safer trajectory on unseen tracks by teaching the agent if is ever uncertain about how to navigate an unseen feature, simply staying in the centre of the track is safe behaviour. However, this is not the most ideal racing behaviour, as this hinders the lap time as the agent prioritises driving in the middle of the track rather than racing. Therefore, we need to adjust the impact of this

penalty to keep the positive effects of the smoother trajectory but encourage it to find a more optimal race line.

In an attempt to find this racing line and counteract this slow progression, a large reward is given to the agent upon completion of a lap. Reward is added based on lap time to encourage fast lap completion. This results in a completion reward that is a function of the lap time L_T, the vehicle's maximum speed, $\bar{v}$ and track length, l_{total}. The lap time bonus r_f is a ratio of the agent's lap time and the fastest possible lap time with the maximum speed

$$r_f = \mathbb{I}(p \geq 1)\left(\frac{l_{total}}{L_T \cdot \bar{v}}\right). \tag{5.14}$$

The fraction in the equation approaches 1 as the vehicle lap time approaches the minimum lap time, therefore, the agent gets a larger bonus for completing a lap in the fastest possible time. Using this arrangement always ensures that at least some positive reward is given on the completion of a lap as none of the terms in the fraction are negative.

These individual rewards are combined with weights w_2 and w_3 to form the final reward function r at each step;

$$r = (p_t - p_{t-1}) \cdot 10 - \mathbb{I}(d_i < d_s) \cdot w_1 - |c| \cdot w_2 + \mathbb{I}(p \geq 1)\left(\frac{l_{total}}{L_T \cdot \bar{v}}\right) \cdot w_3. \tag{5.15}$$

An interesting outcome of the reward function is the agent's ability to perceive the reward from the progress term, despite it not being present in the state vector. Typically, including reward-related terms in the state vector, as with the centring term, helps the agent link rewards to state changes. However, including progress is not feasible for real unseen tracks, as it cannot be recreated. However, the agent seems to infer progress through changes in the LiDAR scans over time, demonstrating the powerful nonlinear mapping ability of DRL networks to extract such nuanced details from raw sensor data.

5.3.1 Reward function weight tuning

The reward function in Equation 5.15 aims to balance the skills required to race in unseen environments while encouraging competitive racing behaviour. The weighting of terms w_1, w_2, and w_3 is aimed at teaching the agent that it should find an optimal trajectory to minimise lap times; however, if the agent is ever uncertain about upcoming features, it can learn that racing in the centre of the track, although not optimal for lap time, will always be the safest trajectory.

As this is the goal, we know that if too much importance is placed on the agent staying centred, it will result in slow and cautious driving behaviour. Conversely, the agent's behaviour could become sporadic and dangerous if too much importance is placed on having a low lap time. This optimal balance between safety and speed must be found

in order to extract the positive driving behaviour that each reward term encourages and minimise unwanted driving behaviour.

To encourage this behaviour, an ideal combination of these weights must be identified. An investigation is conducted to attempt to do this. For this investigation, a range is identified for each weight (1 to 10 for the completion bonus and 0.05 to 0.4 for the centring penalty). An agent is then trained with all possible combinations of these weights, which is repeated to have 10 agents with each combination. Besides the changes in the weights, the agents undergo the same training procedure. Each agent is tested and, based on this performance, a reward is given according to Equation 5.15. The total return received by the agents is normalised based on the total possible reward available given the weight combination.

Figure 5.4 shows a heat map of the average normalised reward for each weight pair. It shows the average return on seen and unseen tracks.

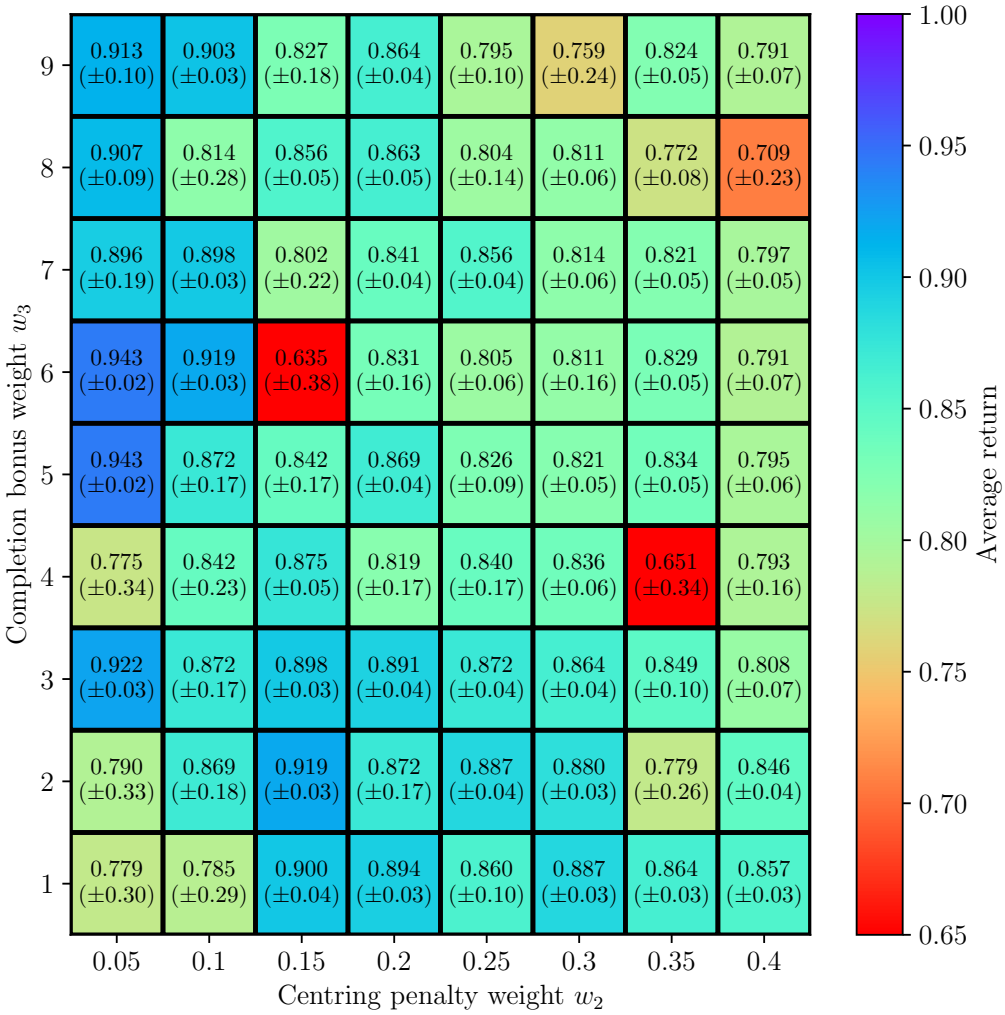


Figure 5.4: The average normalised return and standard deviation (in brackets) of agents with various pairs of reward function weights. The higher values indicate that these agents are able to get more of the overall available reward. This means that these weights helped develop the most optimal policy for this racing problem.

If the average return is closer to 1, it indicates that these agents are maximising the total reward possible and, therefore, have more optimal performance. The heat map shows no strong discernible trend, although some better performing combinations can be identified. The completion bonuses in the upper middle range with lower centring penalties tend to increase performance as the reward is sufficient to encourage faster lap time but not too excessive that it still receives a large reward if the lap time is not optimal. The low centring penalty gives the agent more flexibility to choose a fast race line while encouraging a smooth trajectory. The best performing weights are when the centring penalty is 0.05 and the completion bonus is 5 or 6 indicating that these are the best combination for this reward function.

Notably, it was found that if the centre penalty weight is lower than 0.05, the completion rate of the agent begins to decrease with none of the combinations reliably achieving 100% completion. Consequently, the agent misses out on the completion bonus. Therefore, this is the minimum centring penalty weight considered.

Up to this point only 2 of the 3 weights have been tested; therefore, an investigation should be carried out to examine the effect the crash penalty w_1 has on the performance. The same experiment is repeated, but now only for a subset of the best performing weights, and the crash penalty is altered. Figure 5.5 shows the effect of altering the crash penalty. The previous investigation is carried out with a crash penalty of 1, and it can be seen that increasing this value decreases the agent's performance. Increasing this penalty causes the agent to avoid crashing so much that it prioritises that over the goal of finishing fast and adopts overly cautious driving behaviour. Therefore, a low crash penalty is better as it still incentivises the agent to avoid crashing; however, the other components of the reward function are then more responsible for the racing behaviour of the agent.

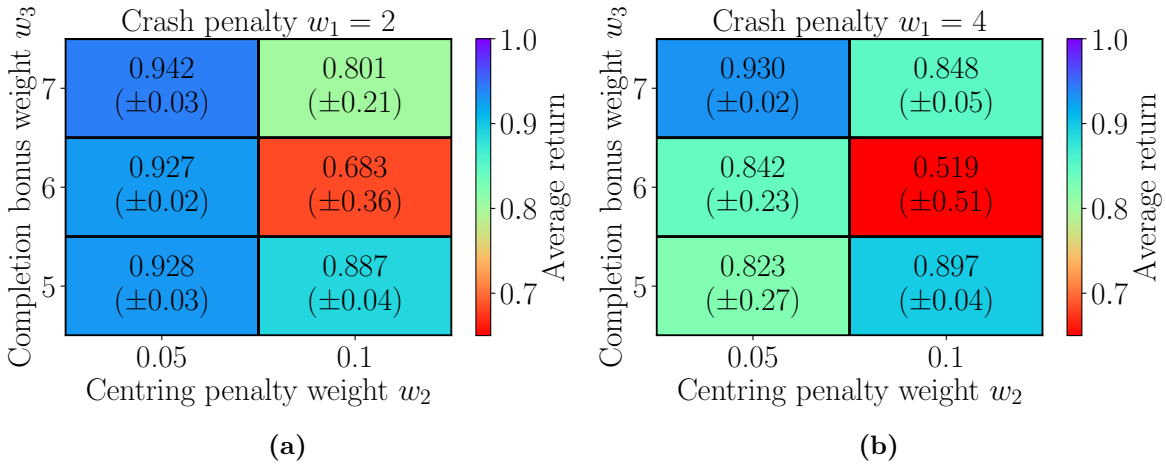


Figure 5.5: Heat maps showing average return for different crash penalty of (a) 2 and (b) 4

The resultant weights used in the reward function are reported in Table 5.1

Parameter	Symbol	Value
Crash penalty	w_1	1
Centring penalty	w_2	0.05
Completion bonus	w_3	5

Table 5.1: Final reward function parameters

5.4 CO-TD3 hyperparameter tuning

TD3 incorporates many techniques that make it a fairly stable and robust algorithm. It is less sensitive to hyperparameter tuning than other reinforcement learning algorithms. However, hyperparameters should still be considered as the environment in which the agent operates and the complexity of the task can require an adjustment in these hyperparameters. The initial hyperparameters proposed by Dankwa and Zheng [69] are used to base the initial hyperparameters on. The actual starting values are shown in Table 5.2 and will be used as the initial values for the network. These are also the values that have been used in the network up to this point. The network is then tuned by altering each of these values while keeping the other values the same as in the table. To obtain a representative average of the results, 10 iterations are trained for 65 000 training steps for each proposed hyperparameter combination.

Parameter	Value
Action noise	0.2
Batch size	100
Discount factor	0.99
Learning rate	1e-3
Policy update frequency	4

Table 5.2: Initial hyperparameters

When the hyperparameters are tuned, the effect they had on the training is examined using the learning curves of each combination. The learning curves are a good representation of the quality and efficiency of training, as they report the return achieved by the agent over the number of training steps. This shows learning progress as the agent undergoes more training. Good training is usually indicated by fastest and more consistent convergence of the return to a maximum value. Figure 5.6 show the learning curve when altering the value for the action noise in the network.

It is clear that the higher noise allowed the agent to learn faster, but it struggled to

converge. The other values showed slower convergence but more stability. A noise value of 0.2 showed a good balance between the training speed and convergence with a small standard deviation toward the end of training.

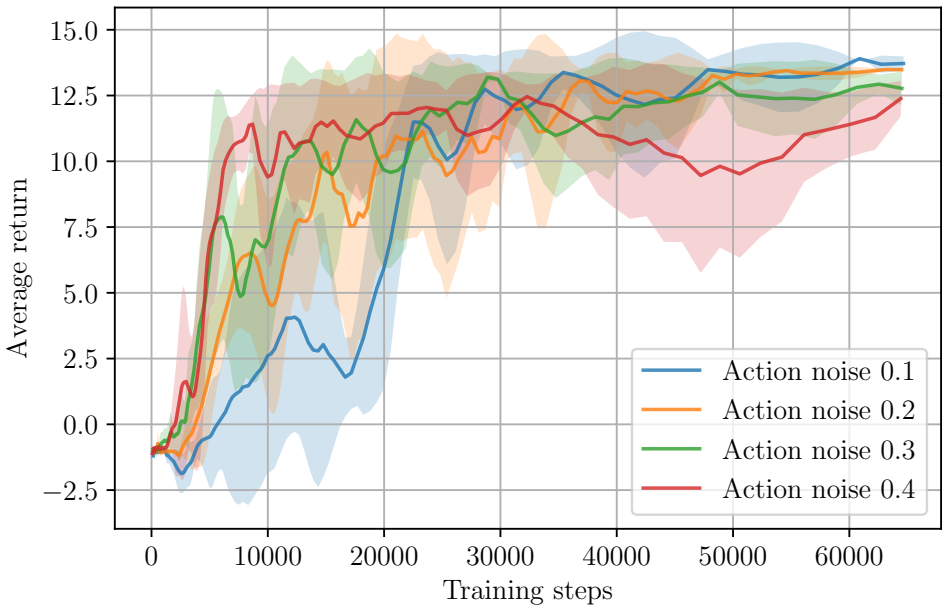


Figure 5.6: The training curve of agents with varying action noise values showing the average return and the standard deviation (shaded)

This process is repeated for the policy update frequency (which describes the number of critic network updates before an actor update), the discount factor (which is the trade off between prioritising short-term versus long term rewards), learning rate (the weight of the parameter updates) and batch size (the number of samples from the buffer). A comprehensive overview of all the hyperparameter tuning is presented in Appendix A. The hyperparameters that have a significant effect on training are the learning rate and action noise, while other hyperparameters, such as batch size, did not have a noticeable effect on the training curve but did increase the training time significantly. The final hyperparameters based on the outcome of the tuning are presented in table Table 5.3

Parameter	**Value**
Action noise	0.2
Batch size	64
Discount factor	0.99
Learning rate	1e-4
Policy update frequency	8

Table 5.3: Final hyperparameters

In addition to the hyperparameter tuning. Appendix A also describes an analysis of

the hyperparameters after tuning. An investigation is conducted that examined the effect of altering just one of the hyperparameters in Table 5.3. The investigation showed that altering any of these hyperparameters adversely affected training, suggesting that the chosen values are suitable to achieve reliable and efficient learning. Furthermore, a similar investigation is conducted on reward function weights. This investigation concluded that even after the hyperparameter tuning, Table 5.1 is still the most effective combination of weights.

5.5 Assessment of state vector variables

The composition of the agent's state vector is carefully designed based on insights from prior literature, theoretical understanding, and practical experience with RL racing agents. Each element, from the number of LiDAR beams to additional vehicle state variables, plays a crucial role in shaping the agent's performance and overall racing behaviour. The effect of these elements can be shown by examining how they contribute to the agents' performance.

Figure 5.7 shows the effect only by varying the number of beams in the LiDAR scan and not any of the terms. Ivanov et al. [35] reported good performance using 21 LiDAR beams, so this is a good indicator of a starting range. The beams are sampled equidistantly from the full 1080-beam LiDAR scan, with an even number of beams chosen to maintain symmetry when calculating the centring term. The results indicate that insufficient scan resolution leads to inconsistent action selection, increasing the likelihood of collisions. The best number of beams was found to be 28, as this provided stable performance. Adding more beams beyond this did not improve performance, but only increased the input size, prolonging training time without additional benefit.

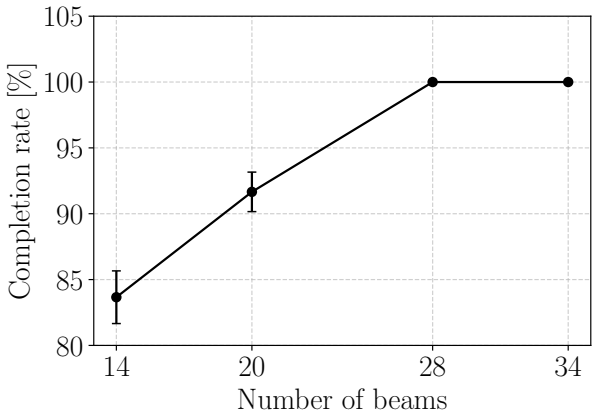


Figure 5.7: The average completion rate of agents with a varying number of beams used in the agents state vector

The impact of including the current speed and steering angle along with the LiDAR scan in the state vector is evident when comparing agents trained with and without these

terms. Removing them leads to a significant drop in performance. Figure 5.8 shows that, without the current steering angle, speed, and centring term in the state vector, agents disproportionately select extreme steering angles, resulting in large and inefficient steering adjustments.

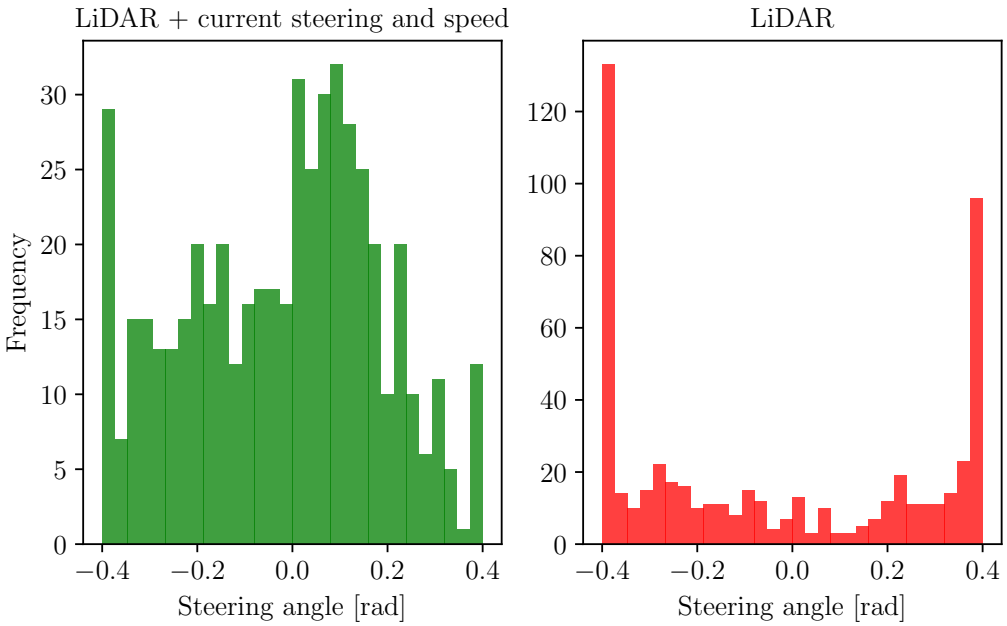
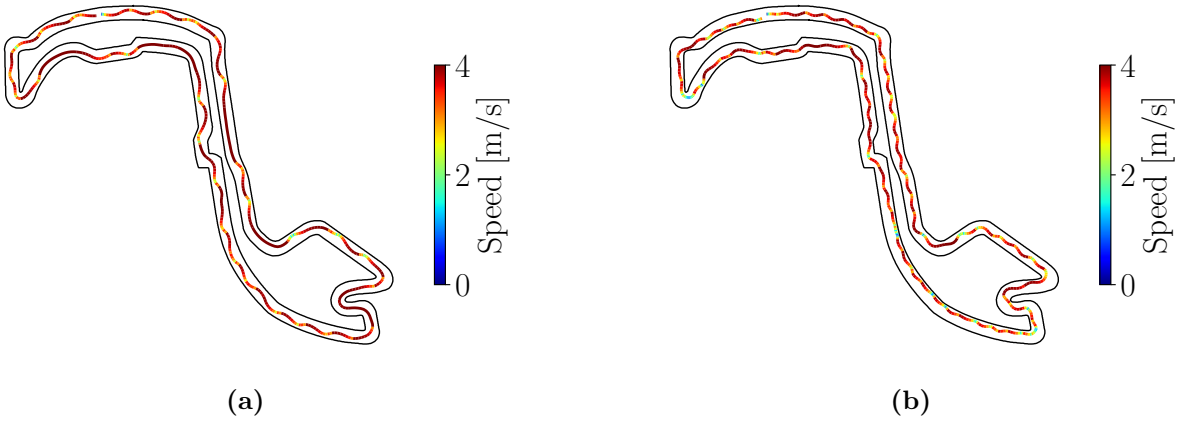


Figure 5.8: A histogram showing an agent's steering angle selection with the current speed and steering angle added to the agent's state vector (left) and when it is removed from the agents state vector (right)

This behaviour not only leads to dangerous driving, but also slows the vehicle down, as excessive steering results in a poor trajectory and reduces speed. In contrast, when the current speed and steering angle are included, the histogram reveals a more balanced distribution. In racing, avoiding unnecessary steering is crucial to maintaining momentum and minimising time loss.

Furthermore, Figure 5.9 shows the trajectory and speed heat map for the same agents as the previous plots. The effect of the poor steering angle can be seen in these trajectories, as the vehicle is constantly slaloming around the whole track compared to the other agent where the trajectory is more consistent. The absence of current speed in the state vector also results in a speed profile with more variations in the speed along the trajectories, as is seen in the vehicle speed heat map.



(a) **(b)**

Figure 5.9: The trajectory heat map showing the path the vehicle travelled the track and the speed a each point. The trajectory of the agent that has the current speed and steering angle in its state vector is shown in (a) and the agent that does not have these terms is shown in (b)

Figure 5.9 shows that although the agent has better performance when the speed and steering angle are included in the state vector, it still has some of the undesired slaloming motion. This is the problem that the centring term directly addresses. Figure 5.10 shows the vehicle's trajectory when this centre reference term is added to the agent's state vector. This improved the agent's action selection by encouraging it to take a smoother trajectory around the track by relying on this centre term for reference and not on one of the boundaries, as it previously did. The smoother trajectory also allowed the agent to maintain higher speeds more consistently as seen in the heat map of the trajectory plot.

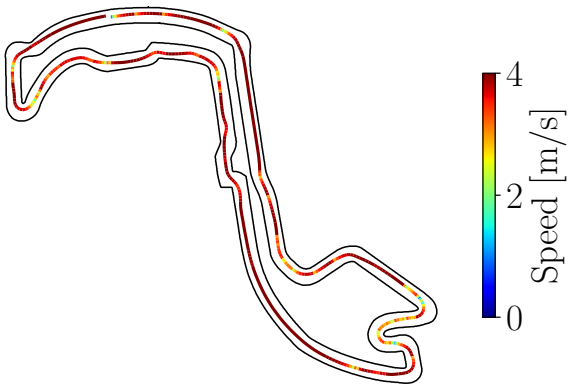


Figure 5.10: The trajectory heat map of an agent with the current speed, steering angle, and centre reference in the agent's state vector

In summary, the design of the agent's state vector and reward function is critical to achieving competitive racing performance. These elements directly influence the agent's ability to select actions that result in effective racing behaviour. When constructing the state vector, it is essential to include relevant data that provide the agent with the necessary

information for decision-making to ensure that the agent can focus on learning meaningful patterns. Additionally, these design choices must consider real-world deployment, meaning that the observations used in training must be realistic and not exploit the simulator's ability to provide all-encompassing state values. In addition, the tuning of both reward function weights and the network hyperparameters are an essential aspect of ensuring the agent can produce the best possible racing performance.

Chapter 6

Training the CO-TD3 racing agent

Now that the specific aspects of the agent's state vector, action vector, and reward function have been defined and optimised, the next step is to establish a training strategy aimed at maximising agent performance. This section outlines the training procedure developed for the agent, with a focus on enabling effective racing on both seen and unseen tracks. Additionally, the chapter evaluates the highest feasible vehicle speed for agent training, identifying the speed at which the agent can still maintain effective control of the vehicle.

6.1 Training strategy formulation

The training strategy describes how agents are exposed to their training environments and what is done to achieve better training efficiency. Initialising the agent at a predefined start point and not changing this throughout the training phase is a common strategy. However, this results in the agent being exposed to certain track features much more frequently, thus decreasing its overall generalisation ability as it develops a bias toward these features.

One of the key challenges we aim to avoid is memorisation. Memorisation refers to the agent's reliance on specific track geometries or sequences of features to select actions, rather than adapting based on its current state [75]. This prevents agents from understanding the actual problem and the dynamics of the environment. We want to avoid the agent's reliance on specific track geometries or feature sequences for action selection, as this undermines its generalisation ability. When the agent overfits the training track, it struggles to generalise to new environments, limiting its performance on unseen tracks. To avoid these pitfalls, the training environment and procedure must be optimised to encourage the development of a robust policy.

To increase the effectiveness of the training on the track, the agent's starting point and racing direction are randomly altered during training. This is done to avoid memorisation of feature sequences and allow for sufficient exploration of the track. The number of episodes per random start point can be adjusted. The optimal attempts before changing the initial position and orientation can be determined by adjusting the number of episodes and examining the effect that it has on the performance. This is shown in Figure 6.1,

where it can be seen that the number of episodes per random start has a minimal effect on the average lap time with only about a 0.4 second range in lap time between agents trained with different number of restarts.

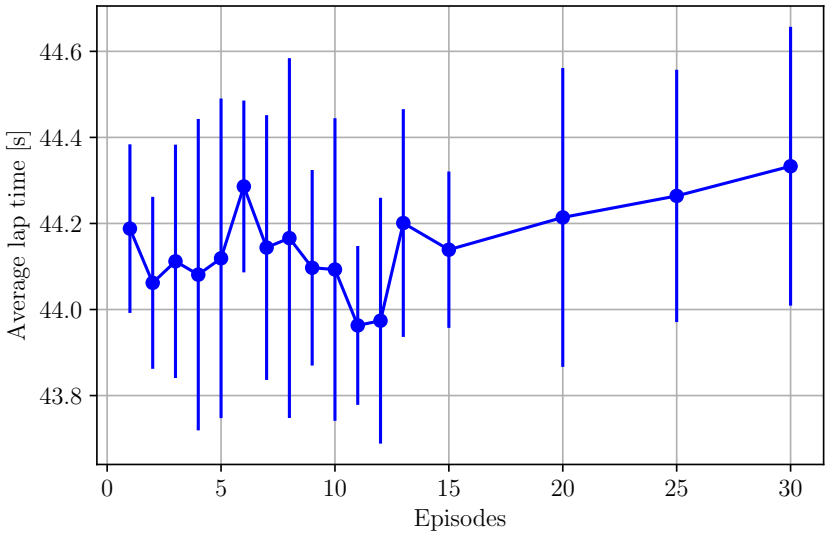


Figure 6.1: Average lap times with standard deviation bars of agents that underwent training with a different number of episodes per random start

It can be seen that the number of episodes that resulted in the lowest average lap time is 11. When examining the training curves for agents, there is no discernible difference between their training efficiency. Therefore, the training procedure is quite robust to this parameter, but 11 will be used as it has the lowest average lap time.

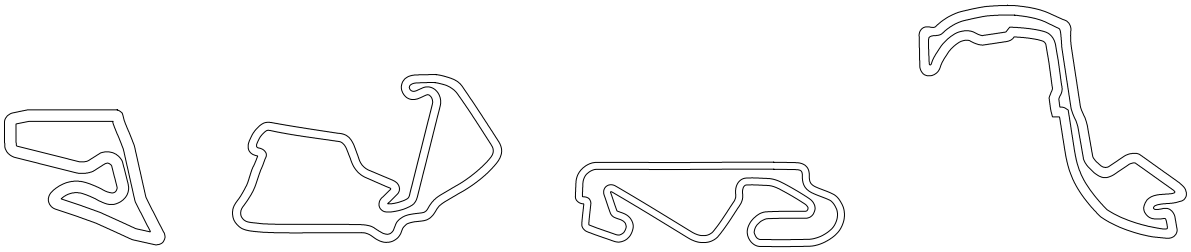
Another factor to consider is whether training on multiple tracks increases the agent's generalisation ability. The effects of this can be determined by selecting a set of tracks to train and test on. A commonly used set of tracks is chosen and the details of the tracks are provided in Table 6.1.

Track Name	Abbr.	Length (m)	Min Curv.	Max Curv.	Avg Curv.
Austria	AUT	94.08	-2.04	0.73	0.187
Great Britain	GBR	200.54	-1.56	1.03	0.147
Spain	ESP	235.08	-1.31	1.20	0.121
Monaco	MCO	176.60	-1.65	1.84	0.176

Table 6.1: Track details including length, minimum, maximum, and average curvature. Negative curvature is convex relative to the track center, while positive curvature is concave.

Additionally, the outlines of each track are shown in Figure 6.2. One group of 10 agents will only be trained on one of these tracks and tested on all the tracks, while the other group of 10 agents will be trained on all four tracks by randomly swapping between

them. The track chosen as the singular track to train on is Monaco. It is selected as it is a challenging track with high average curvature, average length, and various track features.



(a) Austria **(b)** Great Britain **(c)** Spain **(d)** Monaco

Figure 6.2: Track outlines

The training curves in Figure 6.3 show that the agent trains better when the track is not changed as it converges faster and converges with a higher average return. To further investigate whether training on multiple tracks increases its generalisation ability, its racing performance is tested on the four maps. This is compared to the agents solely trained on Monaco and, therefore, the other tracks are unseen.

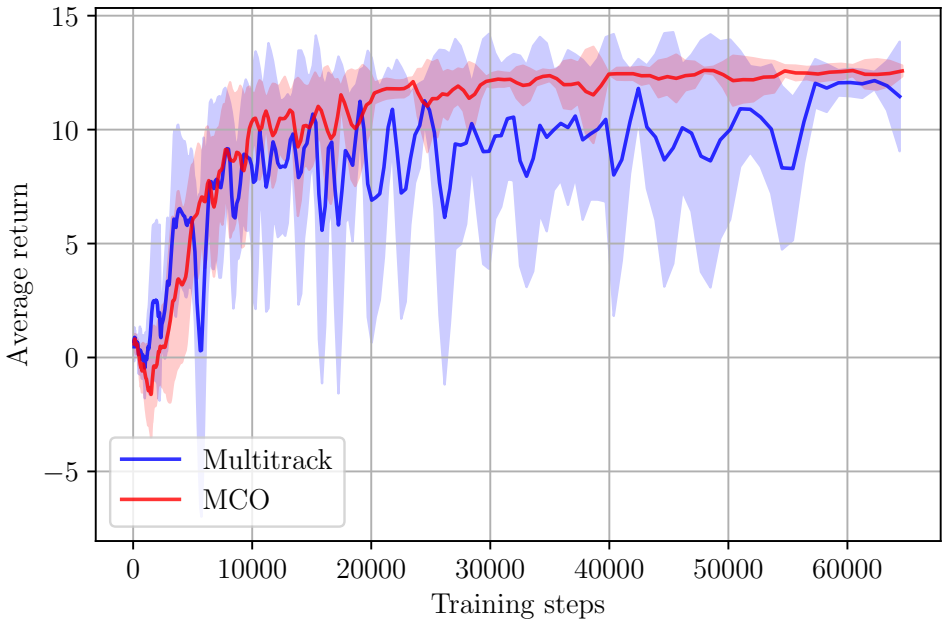


Figure 6.3: The average training curves with standard deviation of multitrack and single track agents

Figure 6.4 indicated that training on a single track produces faster lap times with a smaller standard deviation. This shows that keeping the track constant allows the agent to converge to a more precise policy that has more consistent and superior action selection, which is somewhat counter intuitive. An explanation is that the agent can better learn the effects of its action when the environment remains constant, as it can compare it to

previous tuples from that same environment. This could lead to better Q-value prediction from the critics, which would in turn lead to a more optimal actor network.

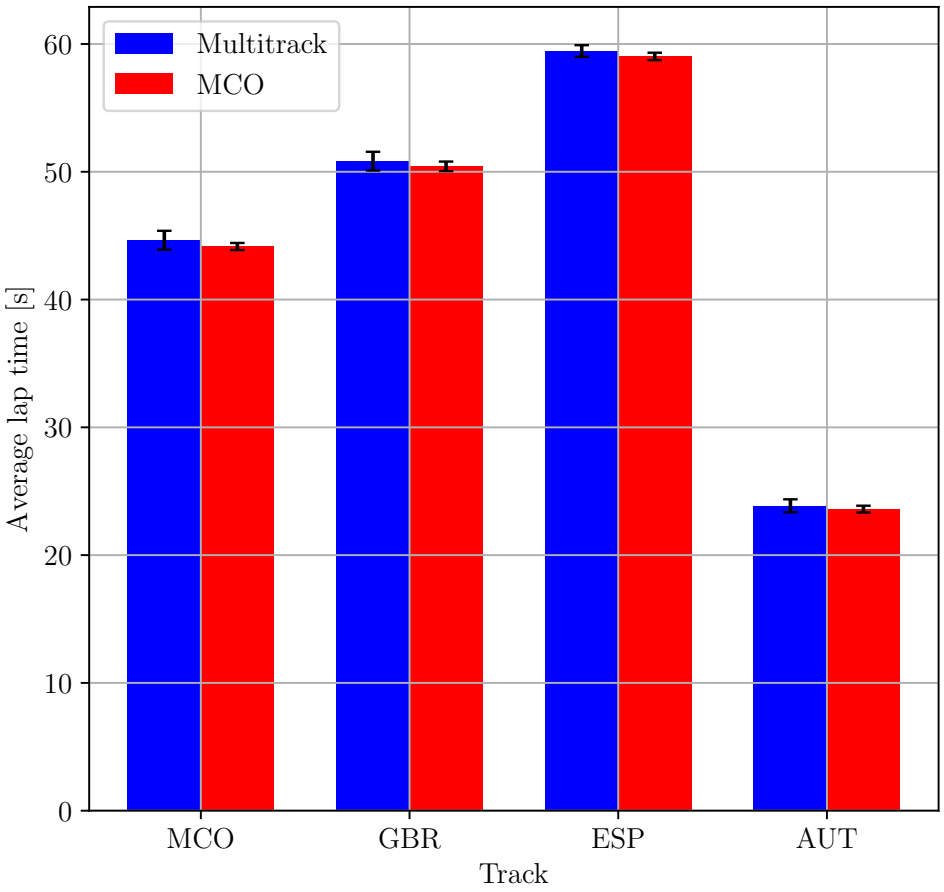


Figure 6.4: The average lap time and standard deviation of multitrack versus single track agents

Notably, the time taken to train an agent capable of this performance given this training strategy is only 672 s. This is quite impressive, as an agent can go from having no racing knowledge to consistently completing laps in just over 11 minutes. [1]

Algorithm 6.1 provides a high level overview of how TD3 is used for the racing problem. Initially, the environment $\mathcal{E}(\mathcal{M})$ is set up with a specific training map $\mathcal{M}$, and the TD3 agent's networks are initialised with the networks described in Section 3.3. For each training step, an action $\mathbf{a_t}$ is selected based on the current policy $\pi_\phi(\mathbf{s_t})$ and exploration noise ϵ, modelled as a Gaussian $\mathcal{N}(0, \sigma)$ to encourage exploration. This action is applied in the environment, giving a reward r_t from the reward function created in Section 5.3 and the next state $\mathbf{s_{t+1}}$, which are stored in a replay buffer $\mathcal{D}$ for stable training.

The TD3 algorithm updates the critic networks Q_{θ_1} and Q_{θ_2} using a mini-batch sampled from the replay buffer to minimise the Temporal Difference (TD) error (Equation 3.19), which progressively refines the value function. The actor network π_ϕ is updated with

[1]The training time was recorded when training on an Intel NUC with a 13th Generation i5-1340P Turbo up to 4.60GHz Processor

deterministic policy gradients to improve the agent's performance in the environment. Random resets, every 11 episodes, introduce random initial states to enhance the agent's robustness across diverse scenarios. This training loop is repeated for T_{steps} steps, allowing the agent to learn an optimal policy through incremental improvements based on both state-action transitions and cumulative rewards.

Algorithm 6.1: Training with TD3

function TRAINTD3
  Input: Training map $\mathcal{M}$, Total training steps T_{steps}
    Initialize environment $\mathcal{E}(\mathcal{M})$ and TD3 agent $(\pi_\phi, Q_{\theta_1}, Q_{\theta'_2}, \pi_{\phi'}, Q_{\theta'_1}, Q_{\theta'_2})$
    Reset($\mathcal{E}$) $\leftarrow$ sample initial state $\mathbf{s_0}$
  for *steps* $= 0$ to T_{step} **do**
      $Episode++$
      Select action $\mathbf{a_t} = \pi_\phi(\mathbf{s_t}) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma)$
      Apply $\mathbf{a_t}$ to environment, observe reward r_t, new state $\mathbf{s_{t+1}}$
      Store transition $(\mathbf{s_t}, \mathbf{a_t}, r_t, \mathbf{s_{t+1}})$ in replay buffer $\mathcal{D}$
      Sample mini-batch from $\mathcal{D}$ and update $Q_{\theta_1}, Q_{\theta_2}$, and target networks
      Update policy network π_ϕ using deterministic policy gradient
    if $\mathbf{s_{t+1}}$ is terminal **then**
      if $Episode\%11 == 0$ **then**
          Reset($\mathcal{E}$) $\leftarrow$ sample random initial state $\mathbf{s_0}$
      else
          Reset($\mathcal{E}$) $\leftarrow$ sample initial state $\mathbf{s_0}$
      end if
    end if
  end for
end function

6.2 Algorithm's maximum vehicle speed

With the algorithm parameters and training finalised, evaluating the maximum speed at which the algorithm can operate effectively becomes an important factor. This evaluation helps determine the speed threshold beyond which the agent's performance may begin to degrade, ensuring reliable and stable control under racing conditions. This affects the algorithm training steps, as more precise control is required at higher speeds. Previous tests were carried out with a maximum speed of 4 m/s and 65,000 training steps, which provided enough steps to show convergence. As the problem becomes more challenging with increasing speed, the number of training steps has been extended to 100,000. It was observed that beyond this point, further training does not yield performance improvements

at any maximum speed. Figure 6.5 illustrates the effect of increasing the maximum speed on the agent training curve. Once again, 10 agents are trained with each maximum speed to obtain the average performance of the agents.

A noticeable observation is a slight decrease in average return as the maximum speed increases. This is an expected consequence of Equation 5.14, as achieving the minimum possible lap time becomes increasingly difficult at higher speeds. As the vehicle approaches its physical limits, the margin for optimal control narrows, making it more challenging for the agent to maintain the same level of performance achieved at lower speeds. Therefore, this trend reflects the inherent challenge of the task rather than a reduction in training efficiency.

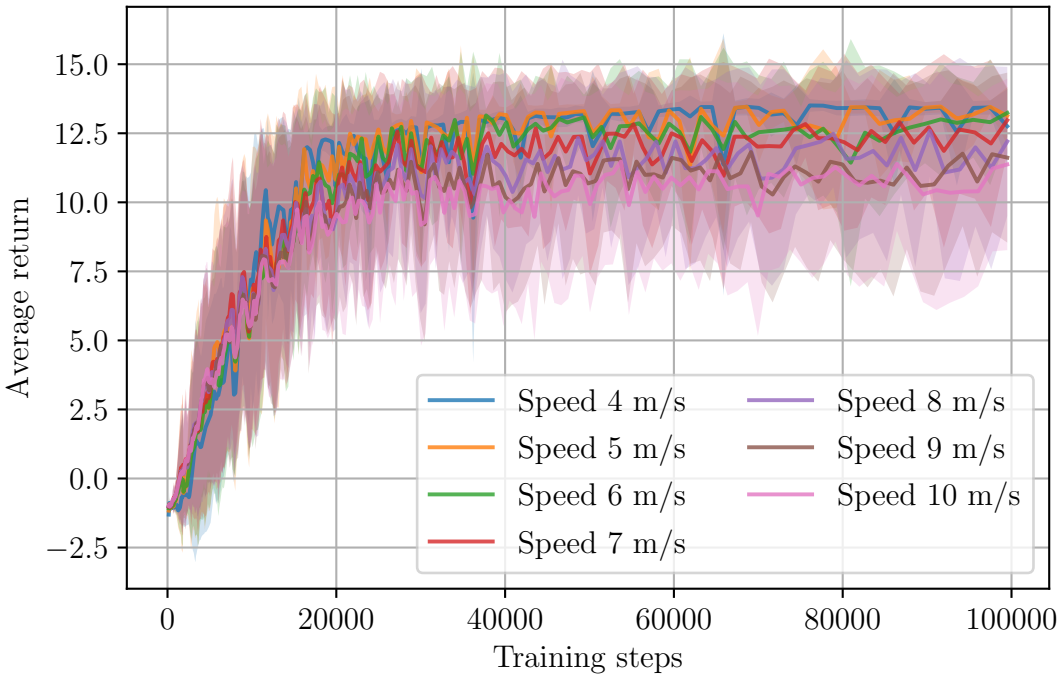


Figure 6.5: The training curve showing the average return and standard deviation (shaded) of agents with different maximum allowable speeds

The curves show a steady increase in return at all speeds as training progresses; therefore, the agent's overall learning is not impacted by the increased speed. However, there is increased variability in the training as the speed increases. Once again, this is expected at higher speeds, as small changes in steering angle have a more profound effect on the overall trajectory. Furthermore, the non-linear response of tyre friction at high speeds now plays a larger role, introducing additional unpredictability and making repeatability more difficult.

To further assess the impact of increasing maximum speed on the agent's racing behaviour, we examine the average lap times achieved across various speed settings. This analysis provides a more direct measure of the effectiveness with which the agent translates its learnt policies into competitive performance. As shown in Figure 6.6, the average lap

times of the agents decrease steadily as the maximum allowable speed increases, following the expected trend. However, beyond a maximum speed of 8 m/s, no advantage in lap time is seen, indicating that this is the maximum speed that the agent learns. In particular, at maximum speeds above 8 m/s, the completion rate suffers, with a higher proportion of agents failing to complete laps.

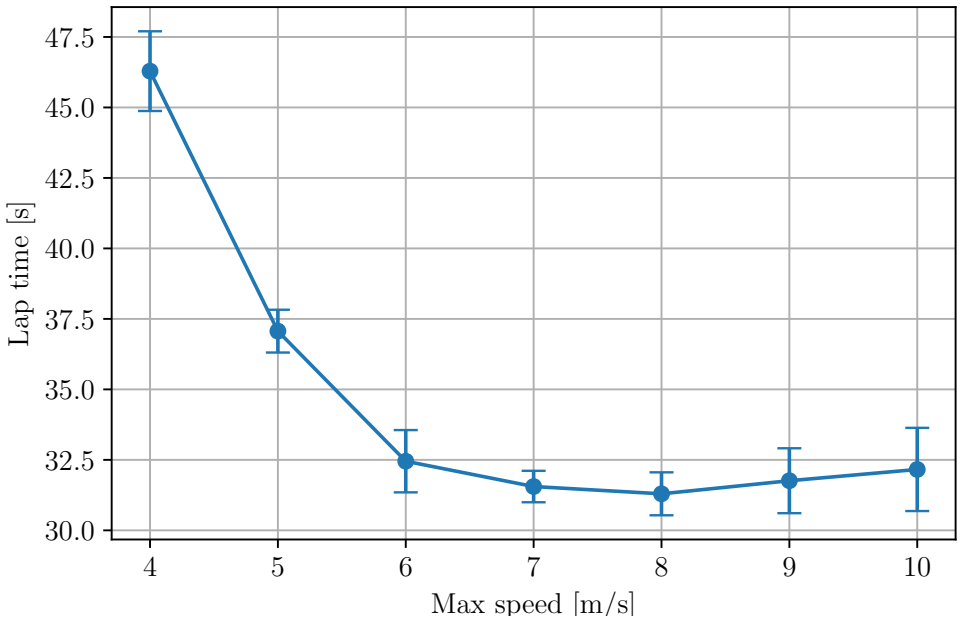


Figure 6.6: The effect of maximum allowable speed on average lap time

This behaviour demonstrates the limitations of the agent's ability to handle the increased speed and maintain control. The observed trends suggest that the agent can perform consistently well up to a maximum speed of 8 m/s, after which the physical constraints of the vehicle and the increased difficulty of the task begin to overwhelm the agent. Consequently, 8 m/s is established as the maximum algorithm speed for reliable performance. The maximum values of the final algorithm are provided in Table 6.2. Notably, with the increase in training steps required for the higher speed, this also results in a longer training time.

Parameter	**Value**
$\overline{\delta}$	0.4 rad
$\overline{v}$	8 m/s
Training steps	100,000
Episodes per random start	11
Training time	1080 s

Table 6.2: Summary of final vehicle and training parameters

In summary, the optimised training approach ensures that the agent can adapt and generalise, directly supporting its ability to navigate both seen and unseen tracks with consistent results. By defining the maximum operational speed, the study also assesses the agent's ability to compete with classic methods as they operate with a similar maximum speed. The ability of our CO-TD3 agent to maintain precise control at these speeds shows its potential to compete with these algorithms.

Chapter 7

Generalisation and random track generator

This section examines the relationship between tracks and the generalisation performance of the agent. The probability of a track being a good track for generalisation is examined. To expand the variety of tracks, a random track generator is implemented, enabling the creation of numerous diverse tracks. This allows for a statistical analysis of the probability that a randomly generated track is good and enables an agent to learn general behaviour.

7.1 Generalisation ability and trained track

The track on which an agent is trained influences the development of the agents' policy depending on what geometric features are present in the track. Intuitively, one might expect that certain tracks, due to their layout, would produce agents with better generalisation capabilities than others. This expectation arises from the idea that training on more diverse or challenging geometric features could better prepare agents for unseen tracks, as it has experienced many different track features. However, the sensitivity to the track on which the agents are trained must be analysed to determine if and to what extent this is true.

To investigate the influence of training tracks on agent generalisation, we use a dataset of 21 F1TENTH tracks modelled after real tracks around the world. The agents are trained on each of these tracks and then tested on the remaining 20 tracks in the data set. The goal is to assess whether agents trained on particular tracks exhibited better generalisation ability across the rest of the tracks. Figure 7.1 illustrates the results of this experiment. It illustrates the frequency of the average completion rate on unseen tracks. The number of successful laps without a collision is recorded and reported as a completion rate across the track set. The completion rates are heavily skewed left, with the highest frequency occurring at the extreme upper bounds. This distribution indicates that, while the training track does influence an agent's ability to generalise, the agents demonstrate a notable degree of robustness. The majority of agents perform reasonably well on different test tracks, suggesting that the influence of the training track, although present, does not

dominate generalisation performance.

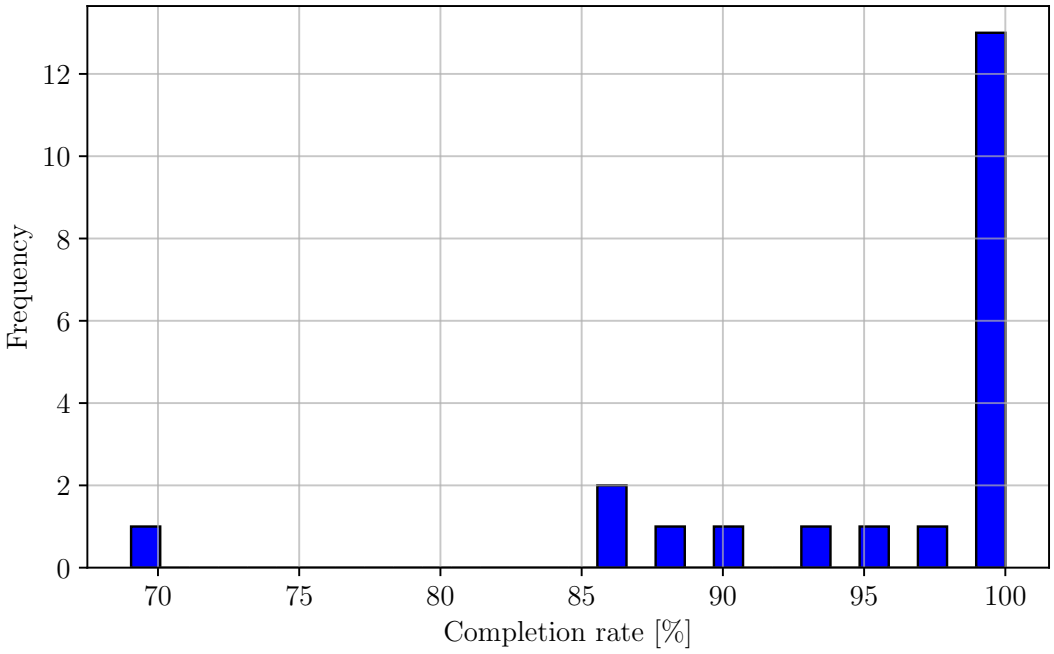


Figure 7.1: The frequency of completion rate of agents tested on unseen tracks

To better understand what makes a track more optimal for training, the specific geometric properties of the tracks were analysed. In particular, a feature that provides insight into the curves and feature diversity of the track is the average absolute curvature of the tracks, the standard deviation of curvature which represents curvature variation, and tortuosity, which is a factor that represents the twistiness of the track. These geometric properties were analysed in order to determine if there is a relationship between these properties and the ability of the tracks to produce agents with *good* generalisation performance. Since measuring the complexity of a track is quite an abstract concept and no standard exists to do this, this analysis did not result in any conclusive relationship between measurable track geometries and completion rate of agent trained on those tracks. Since classifying tracks by some metric is unrealistic, a statistical approach can be taken to determine the probability that a track will be good at producing agents with the ability to generalise. For this a larger track set is needed; with a larger track set we can determine how many tracks to train on to be confident that at least one of the tracks produce an agent with *good* generalisation ability.

7.2 Random track generator

To expand the set of tracks, a large and diverse collection of tracks must be generated. This enables a more thorough evaluation of which tracks yield *good* agent performance and provides information on the probability of achieving such outcomes. A random track

generator is available through F1TENTH [3], however, the tracks that are produced have obscure track geometries that are not realistic. Additionally, this track generator can only produce a few tracks at a time. To address the problems with the previous track generator, a new random track generator is developed. It can quickly and effectively produce tracks with a variety of random shapes and features. This will be useful in expanding the current set of tracks for both training and testing purposes. In addition, the user can choose to produce tracks of random or specific length. This is necessary when training an agent to race in the real world, as the current tracks available in simulation are for large-scale testing with lengths usually exceeding 100 m. This is not a good representation of tracks that the agent would encounter in the real world as F1TENTH tracks are normally much shorter due to practical limitations, such as space availability and the cost of setting up real tracks. Therefore, the track generator can produce tracks in order to prepare the agent for these real tracks. This section outlines the process of how the random tracks are generated.

7.2.1 Basic outline

The first stage generates the basic outline of the track. To achieve a wide range of base shapes, the outline is constructed by generating two shapes. The shapes are randomly selected between an oval, a quadrilateral, and a pentagon. The size of these shapes is randomly chosen within predefined bounds. The size of the oval is dictated by the lengths of the major and minor axes, whereas the length of the sides dictates the size of the rectangle and pentagon. These shapes are then randomly rotated and placed with a random offset from each other's centres, as seen in Figure 7.2.

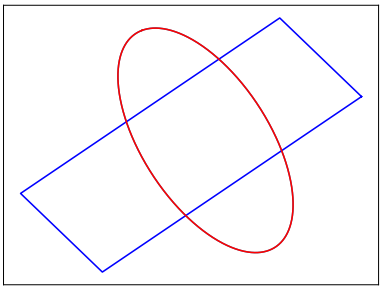


Figure 7.2: The two randomly selected shapes, quadrilateral (blue) and oval (red), are plotted

Once this has been completed, the points of intersection between the shapes are found. The first and second shapes can be defined as the set $\mathcal{A}$ and $\mathcal{B}$, respectively. Having the shapes in this form allows the intersecting set to be found by

$$\mathcal{I} = \mathcal{A} \cap \mathcal{B}. \tag{7.1}$$

The Intersecting set can be filtered out thus, creating the outline set

$$\mathcal{O} = (\mathcal{A} + \mathcal{B}) - \mathcal{I}. \tag{7.2}$$

After this process, the resulting shape is the outline between the two shapes as shown in Figure 7.3. This outline is used as the base centreline.

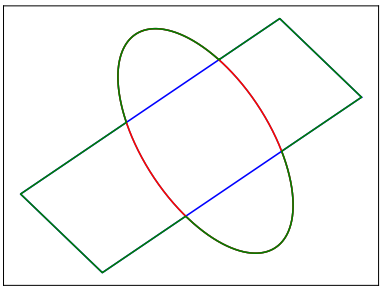


Figure 7.3: The green line shows the outline of the intersecting shapes

7.2.2 Deformation

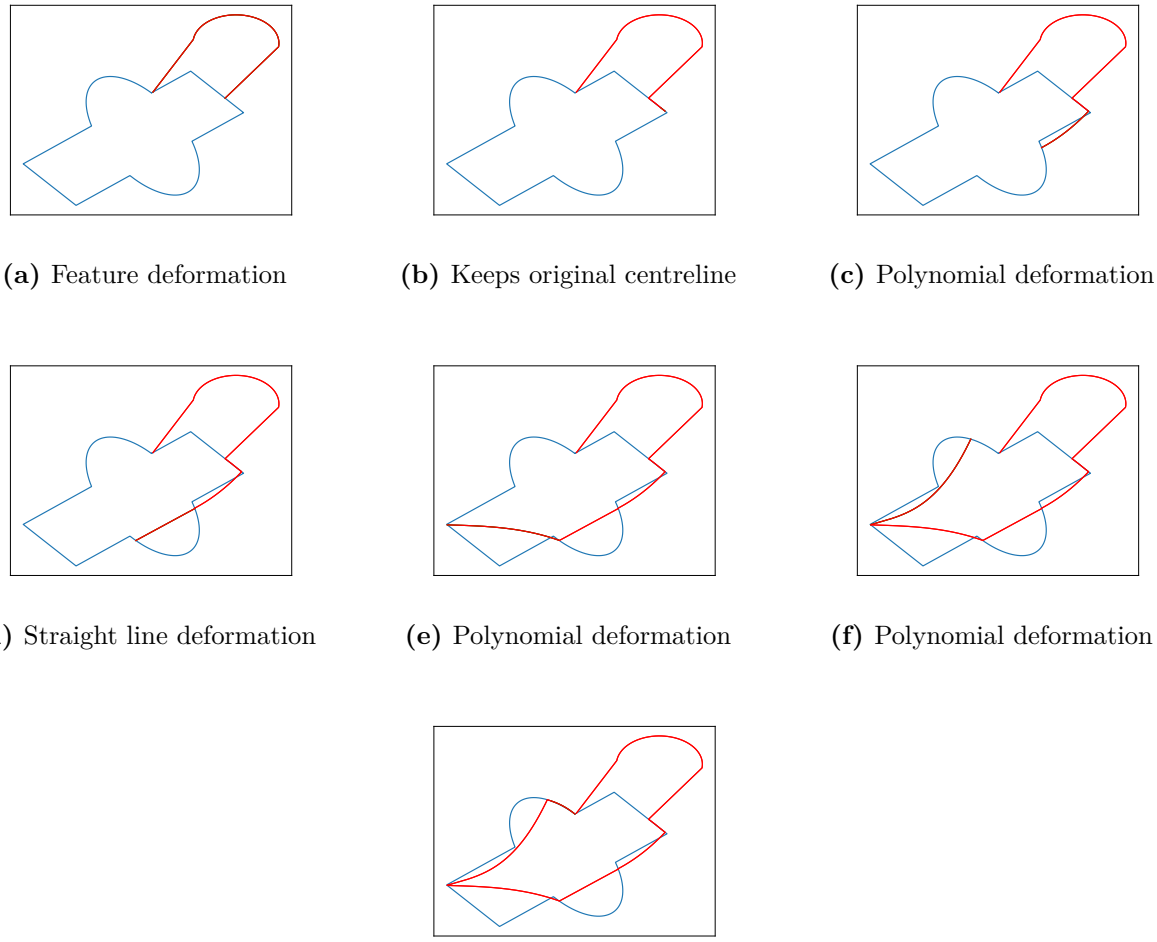
The next step in the process is to deform this centreline in order to create diverse and challenging track shapes. To achieve this, a random number of points are chosen from the centre line. These points will be used as the starting and end points for the current deformation. The type of deformation is randomly selected based on the probability of each deformation feature. The available features are straight lines, polynomials with a random degree, curved features, sharp features, and, lastly, no deformation which will leave the current centreline unchanged. The probabilities of the next feature changes based on the previous one in order to minimise the chances of the track consisting of just one feature.

The straight line deformation is created by simply joining the beginning and end points of the deformation with a straight line. When creating the polynomial deformation, the start and end points are used with the polyfit function to create a polynomial of a random degree. This works by fitting polynomials to data points by minimising the error between the data and the polynomial's predictions. By specifying the degree of the polynomial, polyfit finds coefficients that best approximate the relationship between the input variables. This enables a wide diversity of shapes in the track. The random polynomial function is then used to plot the deformed point in place of the previous points.

The curved feature is used to create unique features on the track that are typical of man-made racetracks. It begins at the start of the deformation section, where a straight line is created that protrudes from the track at a randomly chosen angle that centres around 90 degrees. The length of this protrusion is randomly selected. Once this is created,

a semi-circle is generated that has a random radius. The end of the semicircle is then connected with either a straight or angled line to the endpoint in the deformation section.

The sharp feature is similar to the curved feature; however, instead of the circular section at the end, a sharp feature is created at an angle to form a corner towards the end point of that deformation section. This process ensures that the track has a variety of features and high variability between different tracks. The deformation can be seen in Figure 7.4.



(a) Feature deformation **(b)** Keeps original centreline **(c)** Polynomial deformation

(d) Straight line deformation **(e)** Polynomial deformation **(f)** Polynomial deformation

(g) Keep original centreline

Figure 7.4: The sub-figures show the step-wise deformation (red) of the track by deforming it with different deformation methods that result in a completed track

Once the shape has fully deformed, any problems that could have occurred during the deformation stage are checked. These problems are typically self-intersecting lines on the track. If any of these are present, they will be isolated and removed to ensure that the track remains smooth as seen in Figure 7.5. If they cannot be removed, the track will not be created. The new centre line is saved with all the added features. This process is shown in Algorithm 7.1.

(a) (b)

Figure 7.5: An exaggerated example of (a) a track that has multiple self-intercepting points and (b) the resultant shape of the track after the points in between the intersection are removed and the track is smoothed to show the final track

Algorithm 7.1: Deform centre line

Input: ***centre_line***
$first\_deformation$ = **TRUE**
$end\_point = 0$
while $end\_point <$ LENGTH(***centre_line***) **do**
    $deformation\_length$ = SELECTRANDOMINT(0, $\frac{\text{LENGTH}(\boldsymbol{centre\_line})}{6}$)
    if $first\_deformation$ **then**
        $start\_point = 0$
        $end\_point = deformation\_length$
        $first\_deformation$ = **FALSE**
    else
        $start\_point = end\_point$
        $end\_point = end\_point + deformation\_length$
        if $end\_point >$ LENGTH(***centre_line***) **then**
            end_point = LENGTH(***centre_line***)
        end if
    end if
    $deformation\_type$ = SELECTDEFORMATIONTYPE($previous\_deformation$)
    new_segment = CREATEDEFORMATION($deformation\_type, start\_point, end\_point$)
    centre_line = REPLACESEGMENT(***centre_line***, ***new_segment***)
end while
return ***centre_line***

7.2.3 Pre-processing the centre line

At this stage the track has all the features of the final track, however, the centre line must be processed to prepare it to be converted to the track. If the user decides to make the track a specific length, it should be scaled to the correct length before any other processing step so as not to have any unfavourable effects on the track's features and width. The current centre line will be scaled up or down by multiplying the centre line by a scaling factor sf based on the desired track length l_d and the current track length l_c where

$$sf = \frac{l_d}{l_c} \tag{7.3}$$

In order to do this, the current length has to be calculated. This is done by stepping through each coordinate pair in the centre line and calculating the distance between the current point and the next point. Each distance is then added to the cumulative length of the track until every point has been used and the total length of the track is found. This can be expressed as follows

$$l_c = \sum_{i=1}^{n} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}. \tag{7.4}$$

As the centreline points used in this calculation are in close proximity to each other, it is a viable assumption to simply calculate the straight-line distance between them, and it is not necessary to account for any curvature.

The next step is to ensure that the centre line falls on the origin (0,0), as this will allow the track to be orientated more accurately in an xy plane. To do this, the point closest to the origin is identified. The whole track is then shifted so that the identified point will now fall on the origin. A spline function is then fitted to the centreline to ensure that it is continuous. This spline function also allows the centerline to be slightly smoothed, as seen in Figure 7.6. This will result in a race track that has smooth transitions between the various deformed sections, which will produce a more cohesive track that has natural flow.

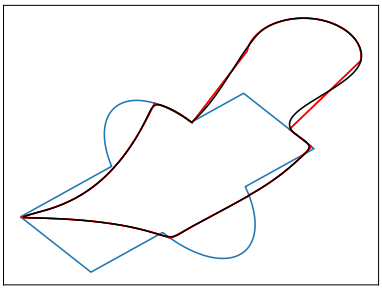


Figure 7.6: The original centreline (blue) the deformed centreline (red) and the smoothed centreline (black) are shown. The rounded corners and smooth transitions between deformation segments show the effects of the spline fitting and smoothing

7.2.4 Generating the track

The centre line can now be used to create the actual track. In order to do this, a buffer of coordinates has to be created at a specified distance on either side of the centre line, which will create the track width. This is done by specifying a track width and generating inner and outer boundaries according to this width. This provides all the information required to create the complete track. The track can now be saved to be used in the F1TENTH

environment. The F1TENTH environment requires the centre line in a CSV file along with a track outline PNG and a YAML file with the name of the PNG, the specific resolution, and the origin of the track.

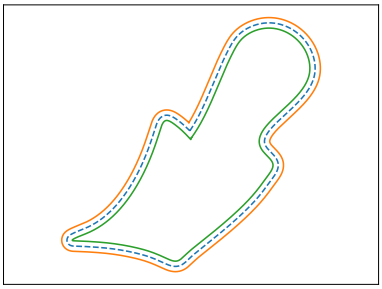


Figure 7.7: The completed track showing the outer bounds (orange) and inner bounds (green)

Figure 7.8 shows a flow diagram illustrating an overview of the entire generation process, from finding the interaction of the random shapes to the final track. Figure 7.9 shows some more examples of the track generated using this random track generator. Figure 7.9 shows that tracks of various lengths, curvatures, and complexities can be created using our random track generator. This is able to provide a divers track set to test the agents generalisation ability by being able to train and test them on more diverse tracks.[1]

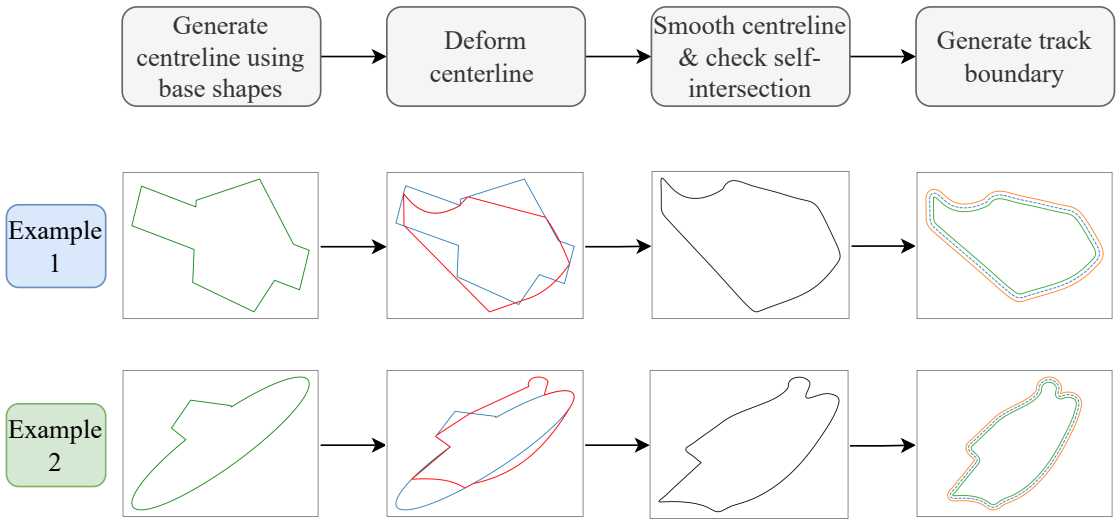
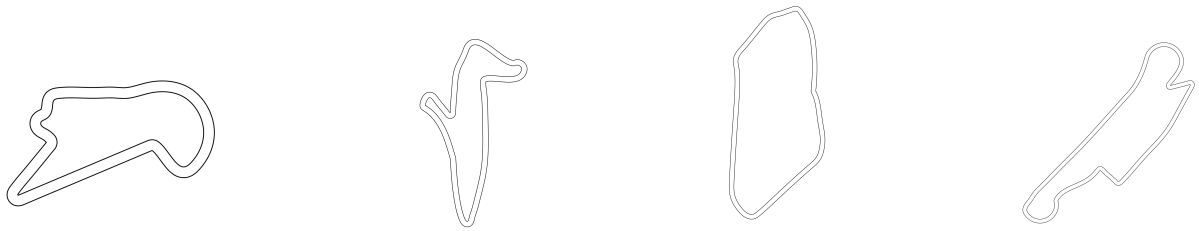


Figure 7.8: The flow diagram indicates 2 examples of the main events when generating a track. It first shows the outline found by finding the intersection of two random shapes. Next this outline is deformed using the deformation methods available. Next the track is smoothed, and lastly the inner and outer boundary is created to form the complete track

[1] https://github.com/D-Jefferies/F1TENTH-Racetrack-Generator.git



(a) (b) (c) (d)

Figure 7.9: Outlines of tracks created with the random track generator

7.3 Generalisation on randomly generated tracks

This section investigates the generalisation ability of agents trained on randomly generated tracks, which is critical to their performance in unseen environments. To evaluate how tracks effect generalisation ability, a set of 100 randomly generated tracks is created. To ensure an accurate representation of the agents ability, 10 agents are individually trained on each of these tracks and subsequently tested on the remaining tracks in the data set. The completion rate is determined in the same way as with the smaller set of tracks in Section 7.1. Figure 7.10 illustrates the results of this experiment, revealing that the highest frequency of completion rates occurs at the extreme upper bounds. This suggests that many agents were able to navigate unseen tracks effectively after training on the randomly generated tracks. However, the concentration of results at the upper limits also indicates that there are still variations in performance among agents, highlighting the need to further investigate how different track features impact generalisation.

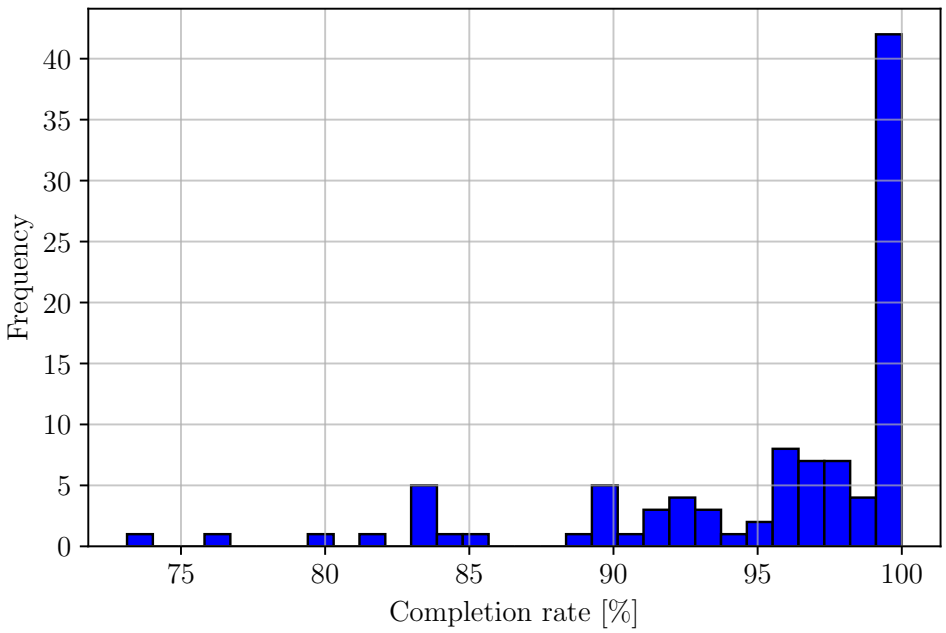


Figure 7.10: The frequency of completion rate of agents tested on randomly generated unseen tracks

The data in Figure 7.10 shows 42 of the 100 tracks where able to produce an agent that can achieve 100% completion. that there is a 42% chance that a randomly generated track will be suitable for generalisation; therefore, there is clearly a distinction between the generalisation of an agent based on the track on which it was trained. So instead of trying to identify one individual track based on some geometric property, we can determine the number of tracks that we need to train on to ensure at least one agent with perfect generalisation ability. This negates the need to have a metric to determine one singular track fit for generalisation. Therefore, we can define the probability of the event (a track that produces a *good* generalisation agent) as

$$P(good\ track) = a = 0.42 \tag{7.5}$$

Based on this, we can determine the probability of generating at least one *good* generalisation track m in a sample of n tracks to be,

$$\begin{aligned} P(m \geq 1) &= 1 - P(m = 0) \\ &= 1 - (1 - a)^n \\ &= 1 - (1 - 0.42)^n \\ &= 1 - (0.58)^n \end{aligned} \tag{7.6}$$

If a certain confidence level P_x is required, the number of tracks required for this can be read off the graph in Figure 7.11 or calculated as

$$n = \frac{\ln(1 - P_x)}{\ln(1 - a)} \tag{7.7}$$

$$n = \frac{\ln(1 - P_x)}{\ln(0.58)} \tag{7.8}$$

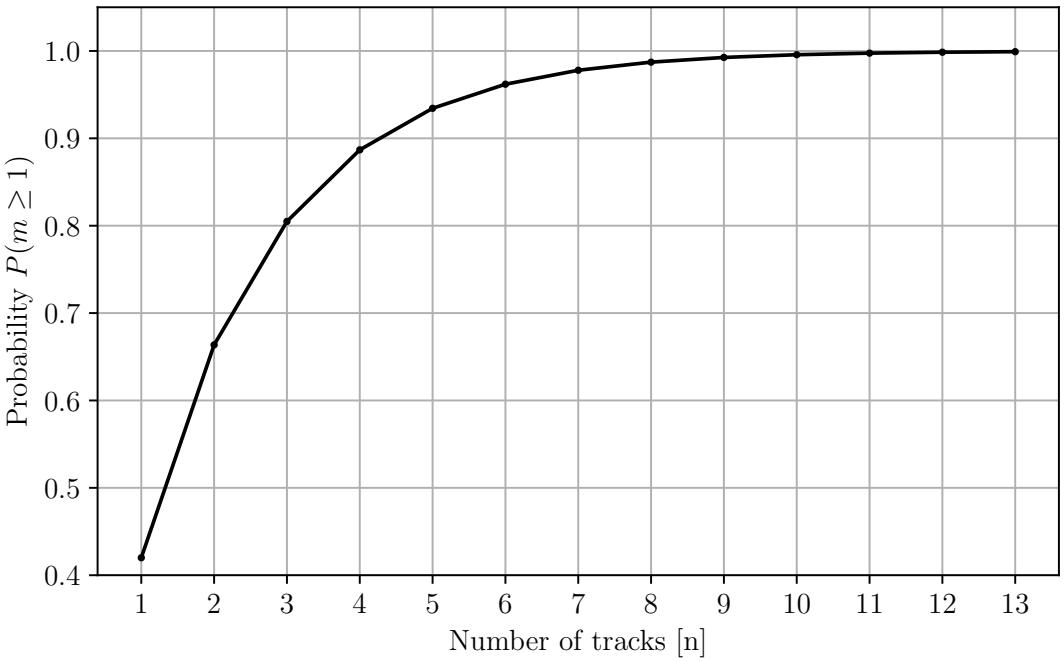


Figure 7.11: The number of tracks required to be produced to ensure a certain probability that one *good* track will be in this set

In summary, his chapter explores the relationship between tracks and the generalisation ability of agents trained in a simulated racing environment on these tracks. As there are no discernible track properties that are able to predict a tracks ability to produce a *good* agent, the only method of ensuring that a *good* generalisation agent is produced is to train on a number of tracks that corresponds to the required confidence level. Therefore, a random track generator is required, which allowed for this analysis and serves as a good tool for testing generalisation and also generating representative track sets for the intended use such as real world F1TENTH tracks.

Chapter 8

Simulation-to-reality transfer problem

The simulation-to-reality (sim-to-real) problem describes unexpected or decreased performance when transferring an algorithm from the simulated environment, in which it was developed, into the real world, where it is intended to be deployed. This section discusses the considerations necessary when developing an algorithm in simulation in order to increase its ability to bridge the sim-to-real gap. The process of transferring a RL agent trained in simulation to the real vehicle is then discussed.

8.1 Transfer process

While the simulator is a valuable tool for developing and comparing autonomous racing frameworks, it does have limitations. Accurately modelling noise in physical systems, such as motors, controllers, and sensors, is challenging. In the F1TENTH simulator, these components behave almost ideally, with controllers showing unrealistically fast response times and no steady-state error or noise. Additionally, this simulator provides information that might not be available or accurately measurable in real hardware, potentially limiting the algorithm's applicability to simulation only if this information is used as an integral part of the design. These challenges highlight the simulation-to-real transfer problem. Considering these limitations is crucial when developing frameworks intended for real-world deployment [76].

The F1TENTH platform uses the robot operating system (ROS) as a communication platform. It works with nodes that are responsible for controlling individual parts of the system. The nodes communicate information to each other via a topic. A topic uses a subscriber and publisher model where one node is subscribed and therefore receives information from the publisher node that sends the information. The user-defined architecture specifies the information and communication between nodes.

The predefined F1TENTH ROS framework uses a mux node that listens to all topics and handles all the communication between the vehicle systems. It communicates with the current framework created by the user and conveys the information from the user node

to the vehicle to drive it as intended by the algorithm. Our implementation to transfer a CO-TD3 agent onto the vehicle uses a two-node system of an agent and a drive node. Figure 8.1 illustrates the nodes and shows the communication between the agent node, the drive node, and the predefined vehicle system that interacts with the environment.

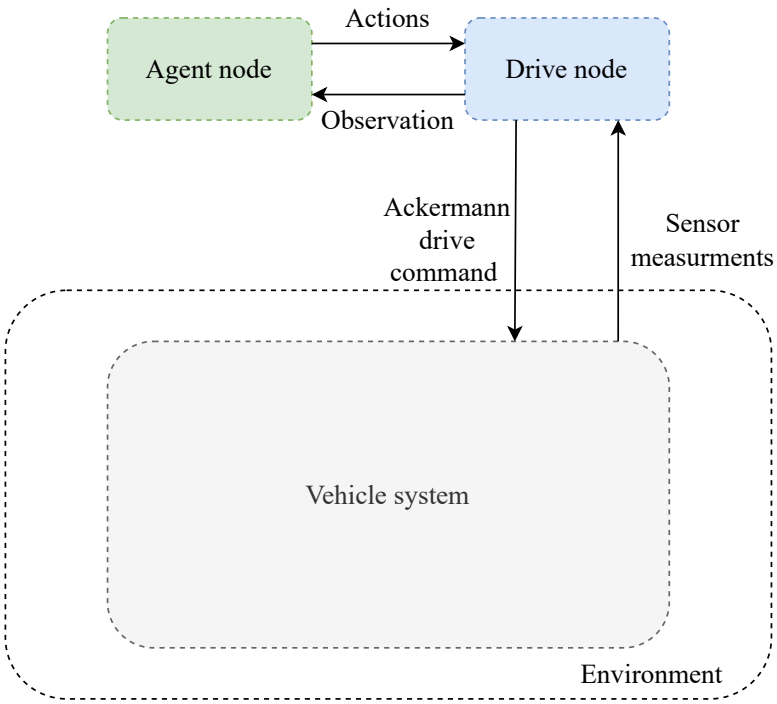


Figure 8.1: ROS nodes with arrows showing the flow of information between the nodes

The drive node communicates with the vehicle sensor topics to get the current LiDAR scan. This scan is then processed in the same way as in simulation by normalising the scan. The drive node then has to recreate the observation as it is in simulation. This is more difficult when using the vehicle, as vehicle states have to be accessed through hardware sensor measurements, which are not always present on the hardware, and if it is present, it is generally more inaccurate than it would be in simulation. This is true when trying to obtain information about the current steering angle and speed. The motor responsible for the speed of the vehicle has feedback; however, it is noisy and does not always mimic the actioned speed due to the hardware delay. This causes the measured speed value to be an inaccurate representation of the actual speed that the agent experienced in training. As for the steering angle, the servos responsible for this do not have feedback; therefore, no measurements can be obtained describing the steering angle.

To overcome these problems, the previous action of the agent can be used as a substitute for the actual speed and steering angle sensor measurement. These mimic the simulated measurements more than the actual measurements, due to the fact that the simulator is such an idealised environment; where there is only a small delay between when an action

is selected and when it is executed. Furthermore, the error between the command actions and the actual actions is normally zero as there is no noise present in the simulated sensor measurements. These measurements are shown in Figure 8.2, where a command is sent for a speed of 3 m/s is shown along the measured speed value in both simulation and the real vehicle.

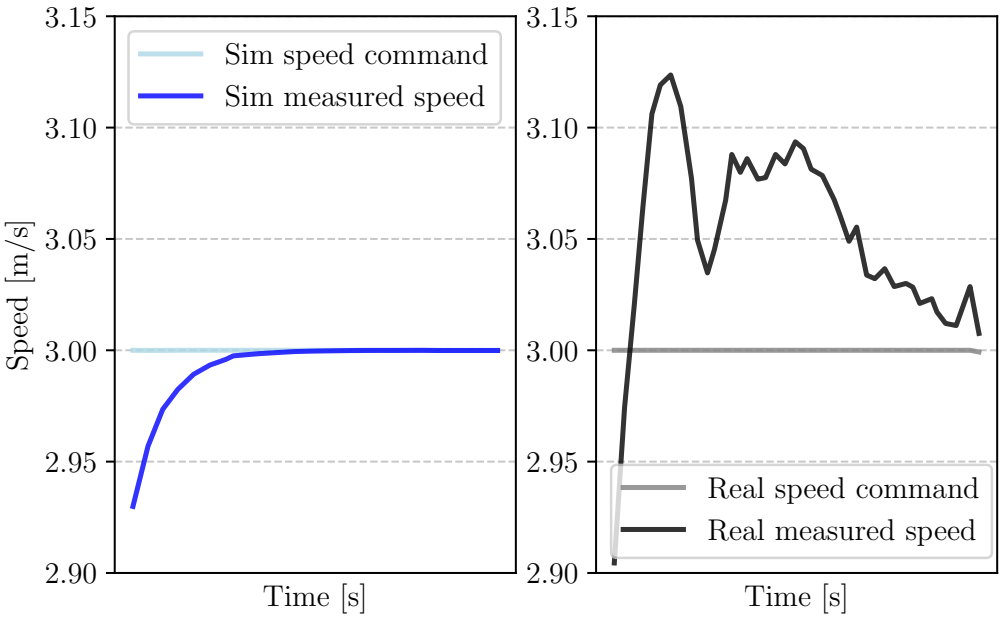


Figure 8.2: The measured speed compared to constant command speed in simulation (left) and in the real vehicle (right) over a time interval of 3 seconds

It can be seen that the real sensor measurement has a lot of error and noise, where the simulated measurement matches the command perfectly without steady-state error after the rise time. This result shows that the previous action is more representative of what the agent receives in training than if it were to be recreated with the sensor measurement. Therefore, the agent state vector is no longer just an observation of sensor measurements but a combination of the processed LiDAR observation, centre term, and the previous action.

$$\mathbf{s_t} = \begin{bmatrix} d_{t,0} & d_{t,1} & \dots & d_{t,n} & c_t & \mathbf{a_{t-1}} \end{bmatrix}^\top, \tag{8.1}$$

This state vector is then used to describe the vehicle in the environment in order for the actor network to select actions. Before the agent node starts receiving observations, it first defines the structure of the neural network. This includes the number of layers and the size of each layer. The network weights are saved after the agent is trained in simulation and loaded in by the agent node from the local file storage on the vehicle. These weights are then initialised into the network structure, thus creating the CO-TD3 agent. The LiDAR observation is then received from the drive node. The previous action as well as the centring term is then added to the agent's state vector. The agent then

selects actions as normal by passing the state to the model actor network,

$$\mathbf{a}_t = \pi_\phi(\mathbf{s}_t)$$

where this action is then scaled based on their maximum values (as in Equation 5.2 and Equation 5.3) before the values are ready to be used as inputs.

These actions are then passed to the drive node, where they are converted to be sent as a drive message to the vehicle system. The vehicle system implements its own controller to convert these inputs into motor inputs.

In order to protect the vehicle, an additional safety feature is designed in the drive node. The drive node continuously checks the measurements in the full LiDAR scan received from the vehicle system nodes and implements an emergency brake if the vehicle gets too close to the boundary or any obstacles. The same collision detection is used as in Equation 4.2, the only difference is that the safety distance d_s is increased to 0.3 m to minimise the chance of a collision that will damage any components.

With this implementation, an RL agent can be trained in the simulated environment and simply be copied over to the vehicle's computer to be deployed on a real race track. The overview of the typical action selection process that takes place in the vehicle is depicted in Figure 8.3, where the colour of the blocks indicates in which node the process occurs, green representing our agent node, blue our drive node, and grey the vehicle system node.

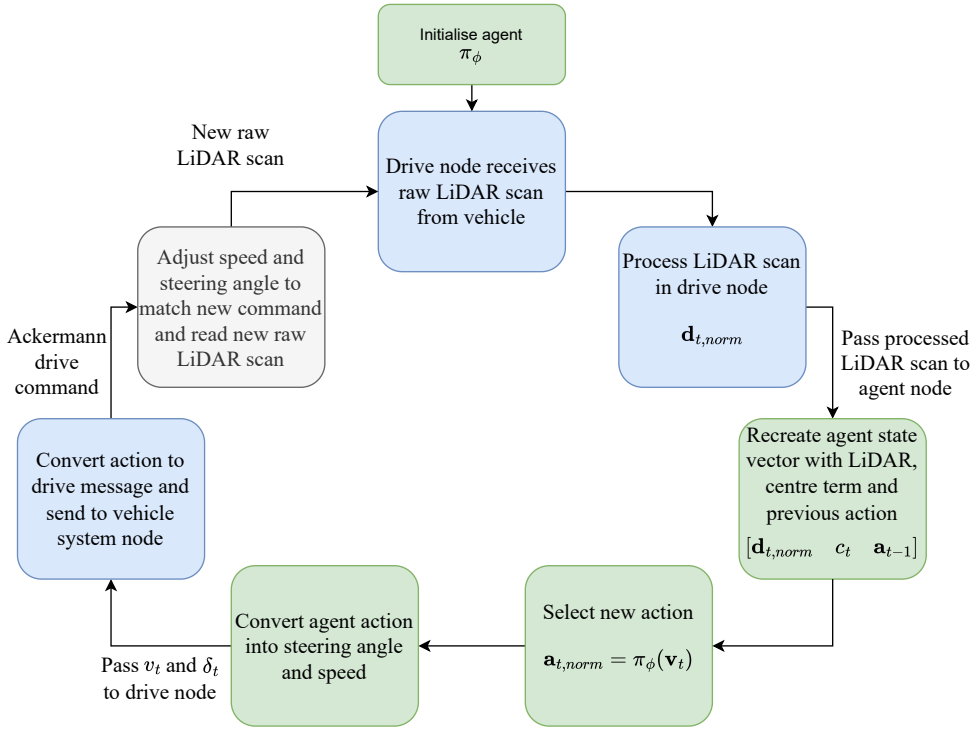
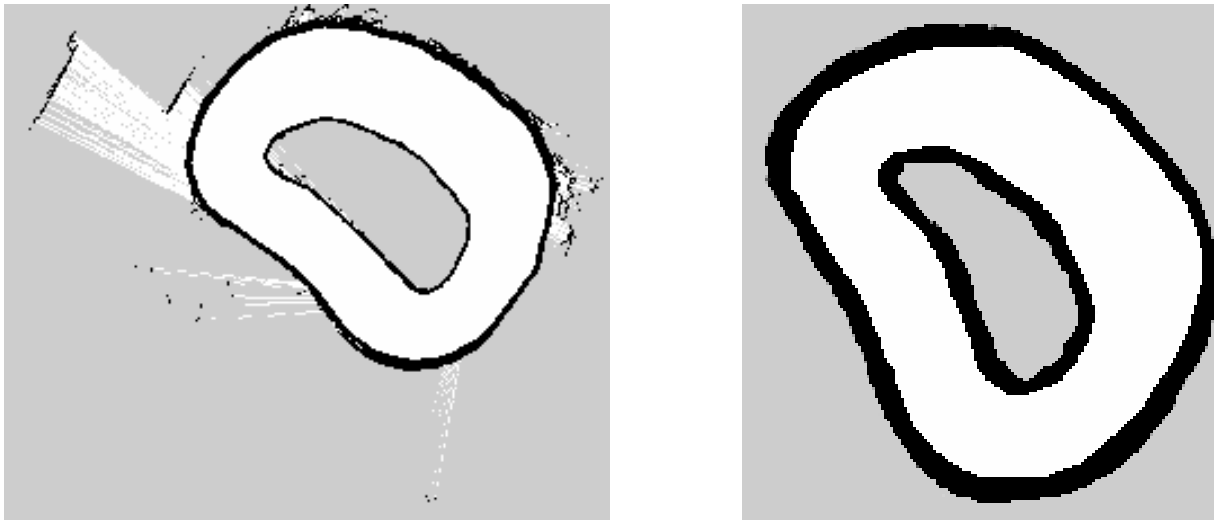


Figure 8.3: The action selection process on the real vehicle

8.2 Mapping and localisation

In order to determine if any performance decrease occurs when deploying the agent on the real vehicle, the physical track should be available in the simulator to train agents as a baseline performance in simulation. Additionally, if the use case calls for the agent to operate on a seen track in the real world, it would still have to undergo training on a simulated version of this track. For this, a map of the track has to be created. This can be done with the LiDAR on the vehicle. The process is performed using a simultaneous localisation and mapping (SLAM) package [77]. This maps out the environment in real time using the LiDAR measurement and can be used with a visualisation tool like RViz to see the progress and map in real time. The map can then be saved and cleaned to be used in the simulator. Figure 8.4 shows a raw map created using SLAM and a cleaned version of that map that removes any noise and makes the boundaries more defined.



(a) **(b)**

Figure 8.4: A comparison between a raw SLAM map (a) with no post-processing and the cleaned version of that same map (b) with the background filled in and the borders thickened in the outwards direction keeping the track footprint constant

In addition to using the map to create a track in the simulated environment, this track will aid in data collection during testing with the real vehicle. Using this track as a reference, the position of the vehicle within the track bounds can be determined using localisation. A particle filter (PF) will be used for localisation. The PF works by estimating a set of poses called particles. After this initial set is determined, the orientation and location of these particles are updated based on data obtained from odometry readings. These readings are then applied to a movement model. The LiDAR sensor readings are then compared with the current environment, and this is used to update the belief of each particle, that is, where the particle is on the track. The particles are then resampled based on the updated belief, and the weighted average of all the particles is used to determine

the position and orientation [78]. The particular particle filter that we used makes use of a compressed directional distance transform (CDDT) to decrease localisation time and computational cost. The CDDT stores the distance from the nearest obstacle at a particular angle θ. The scene geometry is then rotated by θ so that the ray casting is always done at $\theta = 0$. This differs from normal approaches which are not rotated, meaning that the ray casting is done at θ; therefore, it is always done in a different direction. The rotation of the CDDT method allows easier extraction of obstacle points, improving its speed and computational efficiency [78].

The particle filter is used to store the trajectory of the vehicle as it races along the track. The action and current odometry readings are also stored at each step. Using this, similar racing metrics can be obtained from the real vehicle which can be compared directly to those in simulation. To manage data collection, a new ROS node is added to the system. It is triggered by the command sent by the vehicle's ROS node to power the motors. This is because the agent and the drive node begin sending actions as soon as it is initialised; however, these actions are not realised until the user switches the vehicle into autonomous racing mode. Therefore, only once this happens does the data capturing begin. This minimises the post-processing of the data. Additionally, this allows the use of the internal clock to accurately and consistently record lap time. The clock is called when the data capture node begins collecting data, and consequently the time from when this occurs until when the vehicle crosses the finish line is the stored lap time. Figure 8.5 shows how the particle filter and data capture node integrate with the existing system.

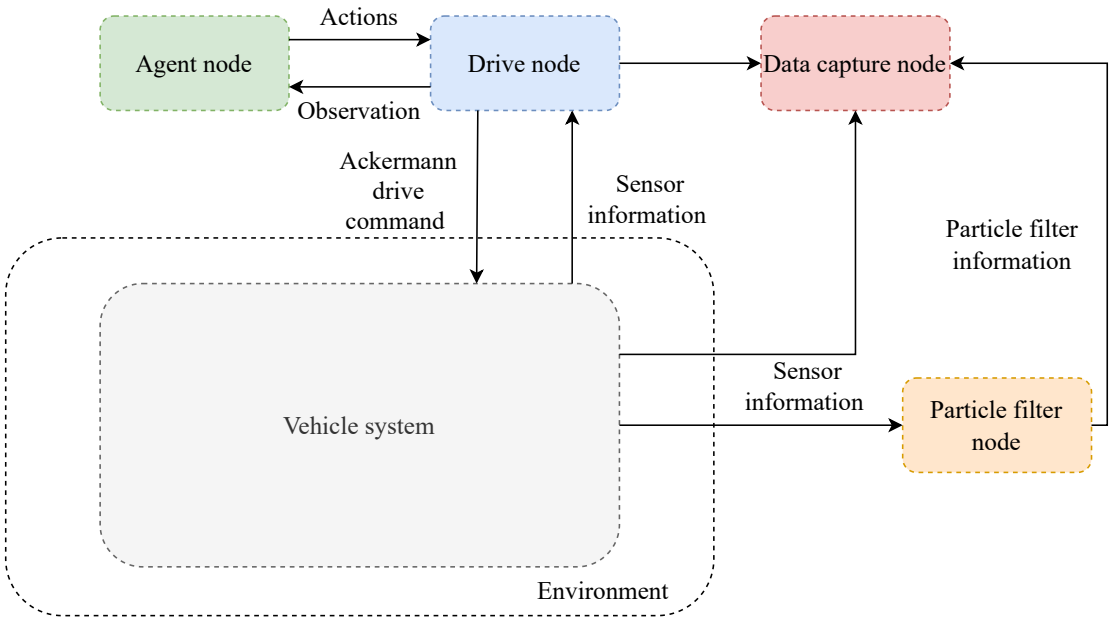


Figure 8.5: ROS nodes structure showing the transfer of information between the nodes

8.3 LiDAR noise model

The simulator attempts to mimic real-world conditions by adding random noise to the LiDAR scans. The noise is added to the scan array before the simulated scan is returned to the agent. The noise is modelled as a Gaussian or normal distribution. For a given LiDAR reading, the noise can be modelled as a random vector $\mathbf{Z}$ added to the true distance $\mathbf{d}$, such that the observed distance $\mathbf{d}_{\text{obs}}$ is

$$\mathbf{d}_{\text{obs}} = \mathbf{d} + \mathbf{Z}. \tag{8.2}$$

Here, $\mathbf{Z}$ is a Gaussian random variable with mean $\mu = 0$ and standard deviation σ_n, i.e., $\mathbf{Z} \sim \mathcal{N}(0, \sigma_n^2)$.

The value for σ_n is rather insignificant when only in simulation (assuming that it is still in a realistic range), as the same LiDAR will be used in both training and testing. The impact of the extent of the noise becomes pertinent when transferring the agent to the real vehicle. The value used for the noise standard deviation should best match the noise present in the LiDAR intended to be used for practical tests. This value is altered in the training phase and tested on the real vehicle to determine the best standard deviation value. The standard deviation ranged between 0 and 0.04, Figure 8.6 shows the effect that noise has on the scans.

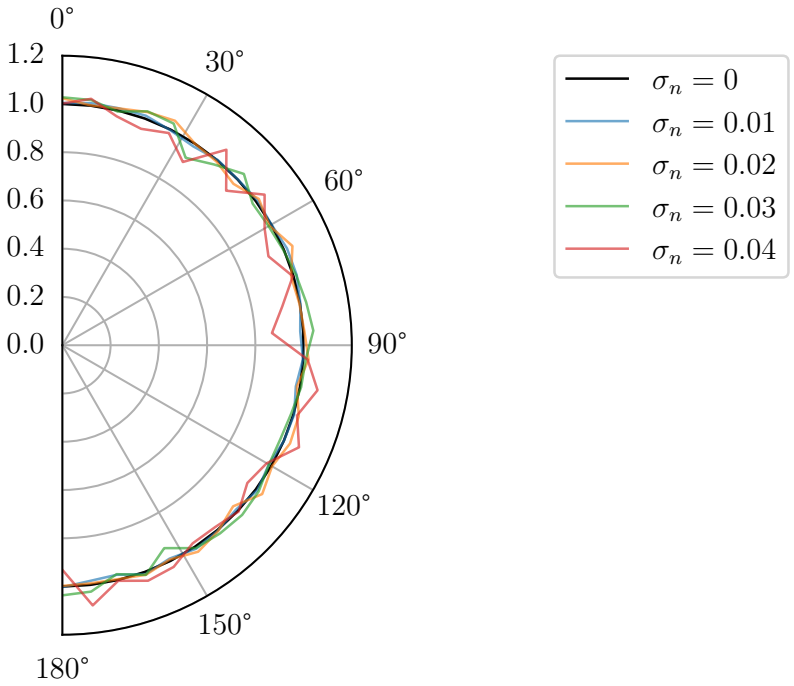
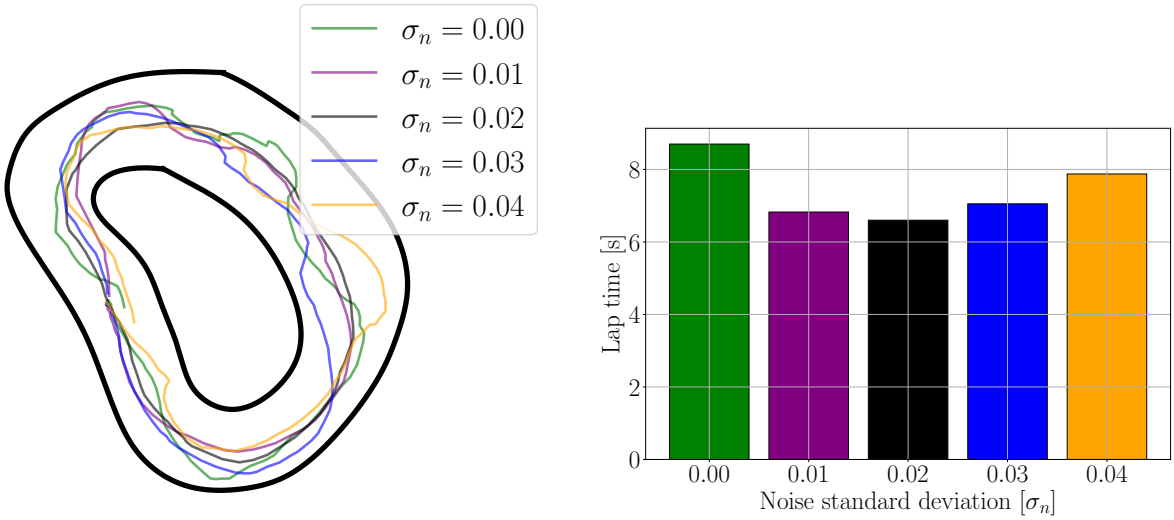


Figure 8.6: A scan reading a constant distance of 1 m with varying values of standard deviation for the Gaussian noise.

Each agent trained on these standard deviations is tested on the real vehicle to assess

the impact on performance. Instead of choosing a noise model based on real scans, which can fluctuate with varying environmental conditions, this approach allows us to account for the variability in dynamic scenarios. Measurement of an accurate noise profile from static tests can be challenging when the intended tests are dynamic. Therefore, we systematically tested a range of values to identify the one that yields the best performance. Figure 8.7 shows the trajectory and lap times for these tests. It can be seen that a σ_n value of 0.02 matches the noise present in the real LiDAR, as it performs much better than the agents with other σ_n values; therefore, this will be the noise used when training the agents in simulation.



(a) **(b)**

Figure 8.7: The trajectories (a) and lap times (b) for agents trained with different σ values. The plots indicate each value's effect on the performance of the vehicle.

The successful transfer of the CO-TD3 racing agent from the simulation to the real world is crucial to its design. By integrating the agent into a ROS network, we demonstrate its practical applicability in real-world racing scenarios. This transfer enables us to test the performance of the agent beyond simulated environments, providing valuable insight into its effectiveness in dynamic real-world conditions.

Chapter 9

Racing performance: Experiments and results

This section discussed the results achieved by our CO-TD3 agents both in simulation and on physical hardware. The results achieved in simulation are presented and a comprehensive comparison between our methods and other existing benchmarks is drawn. The performance of the agent on seen and unseen tracks is reported, and a conclusion about the agent's generalisation ability is made. The performance of the agent on the real vehicle is shown, and a direct comparison between this and the simulated results for the real track is discussed. The agent's ability to generalise on the real vehicle is also reported.

9.1 Simulation

The results obtained in simulation can be used to compare with existing benchmarks and evaluated to identify how our algorithm compares. Furthermore, the degree to which our algorithm can perform in a simulated environment can be tested by assessing its generalisation ability on unseen tracks and with obstacles.

9.1.1 Seen tracks

The performance of our centre-orientated TD3 (CO-TD3) algorithm is evaluated based on commonly reported racing metrics such as lap time and completion rate. Austria (AUT), Spain (ESP), Britain (GBR), and Monaco (MCO) are the four baseline tracks used for this evaluation.

To obtain results for this experiment, 10 agents per test track are trained in simulation. Each of the agents is trained for 100,000 training steps with a maximum allowable speed of 8 m/s. Once training is complete, each agent completes one lap on the track on which it is trained in simulation. The lap time of the fastest agent is reported in Table 9.1 along with the standard deviation for all 10 agents.

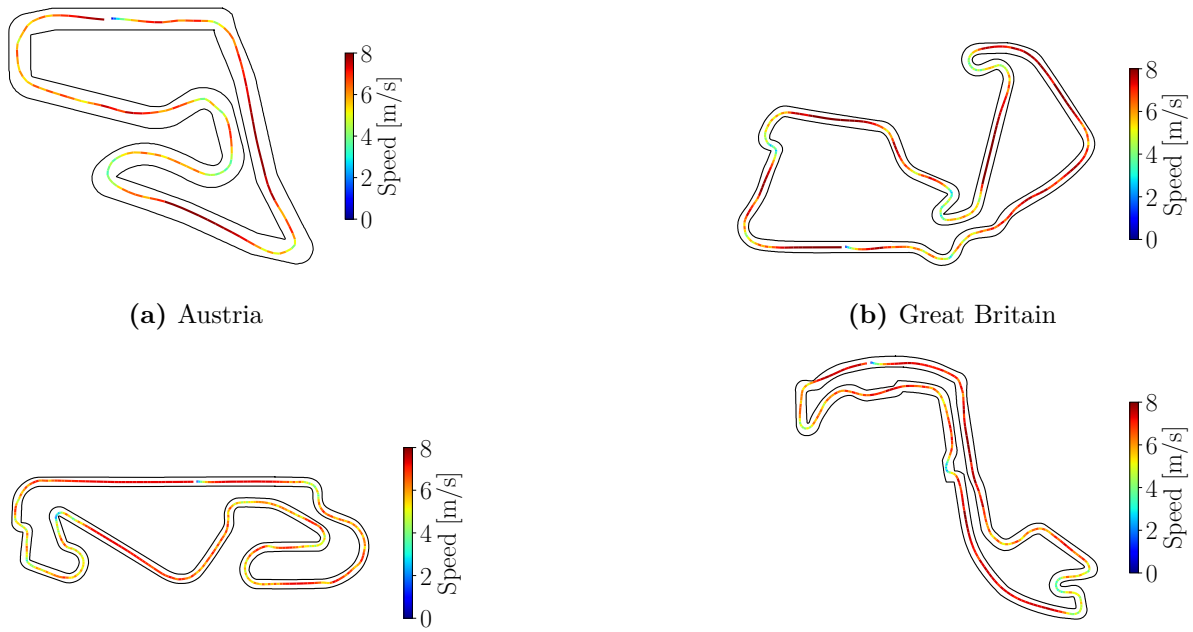
These results are compared to the benchmark results, reported by Evans et al. [71], of other algorithms using the F1TENTH platform. These authors compared three classic control algorithms and one RL algorithm. The classic algorithms are a trajectory optimisa-

tion and tracking algorithm, a model predictive contouring control (MPCC) algorithm, a follow-the-gap algorithm, and a trajectory-aided learning (TAL) end-to-end RL algorithm. Furthermore, we compare the results achieved by Bosello et al. [33] DQN algorithm and Brunnbauer et al. [34] Dreamer + Occupancy (DO) RL algorithm. Table 9.1 shows that our CO-TD3 method performed better than the follow-the-gap, TAL, DO, and DQN methods on all tracks. Furthermore, it outperformed MPCC on AUT, GBR, and MCO. Finally, the agent trained on AUT performed the best beating all other benchmark algorithms on this track. The CO-TD3 algorithm is able to achieve results comparable to state-of-the-art classic control methods, as it is able to maintain precise control over the vehicle at high speeds. This is a significant result as our CO-TD3 is the only end-to-end method to outperform a classic algorithms. The results obtained from our CO-TD3 agents are much closer in performance to classic methods than the other end-to-end methods, which proves that it is a viable option for competitive racing vehicle control.

Planner	**Map**			
	AUT	**ESP**	**GBR**	**MCO**
Opti. & tracking	16.79	**35.92**	**31.24**	**28.08**
MPCC	16.87	39.13	35.40	31.53
Follow-the-gap	19.10	45.78	39.34	34.99
TAL end-to-end	19.94	46.37	40.22	34.93
DQN	∼23	∼56	∼48	∼42
DO RL	∼31	∼72	-	-
CO-TD3	**15.51** ± 0.84	39.38 ± 1.13	32.24 ± 3.21	28.45 ± 1.71

Table 9.1: Mean lap times [s] for optimising and tracking, MPCC, follow-the-gap, end-to-end methods, DQN, DO, and CO-TD3. ∼ indicated that results are obtained from a graph, therefore the best estimate for these values is reported

The trajectory and speed heat map of the agents' laps around each track are shown in Figure 9.1. The smooth trajectories illustrate the agent's capability to maintain consistent control, as evidenced by the lack of any unexpected slaloming or jerking motions. This smoothness in the trajectory and competitive lap times indicates that the agents have an effective racing policy, where there is a balance between speed and stability while navigating the track. Additionally, the speed heat map highlights how the agent adjusts its speed in response to different track segments, such as slowing down for sharp turns or accelerating on straights. The trajectories highlight these aspects of the agents racing that allows it to compete with classic methods such as trajectory optimisation and MPCC.



(a) Austria

(b) Great Britain

(c) Spain

(d) Monaco

Figure 9.1: The trajectories with speed heat map show the path the agents took around each of the test tracks to achieve the lap times reported in Table 9.1

9.1.2 Unseen tracks

The key factor that distinguishes RL racing algorithms from classic methods is their generalisation ability. We present results that show that our CO-TD3 agents can generalise to navigate unseen tracks safely and that the agent's racing policy can transfer to unseen tracks.

The generalisation ability of our CO-TD3 agent is evaluated by recording its ability to complete laps on unseen tracks. To generate these results, agents are trained on the MCO track, using the same method as the seen tests, and tested on a randomly generated set of tracks. MCO is chosen as the track to train these agents as it is a complex track with a variety of challenging features and an average track length. Our random track generation is used to create the set of unseen tracks. The generalisation ability of the CO-TD3 agent is compared to the generalisation ability of a standard TD3 agent. The standard TD3 agent underwent the same training strategy. However, the agent's state vector consisted of just an observation containing the full LiDAR scan. The centre penalty and the completion bonus was also removed from the reward function.

Algorithm	Number of unseen tracks tested on	Completion rate	Max speed [m/s]
TD3	50	46.67%	4
CO-TD3	50	100%	8

Table 9.2: Generalisation performance of agents trained on MCO

The CO-TD3 agent performed much better, with a perfect completion rate. The regular TD3 agent could not achieve constant performance even at lower speeds as the speed had to be limited to prevent a 0% completion rate. This shows the benefits of having additional features in the agent's state space and the effects of the improved reward function.

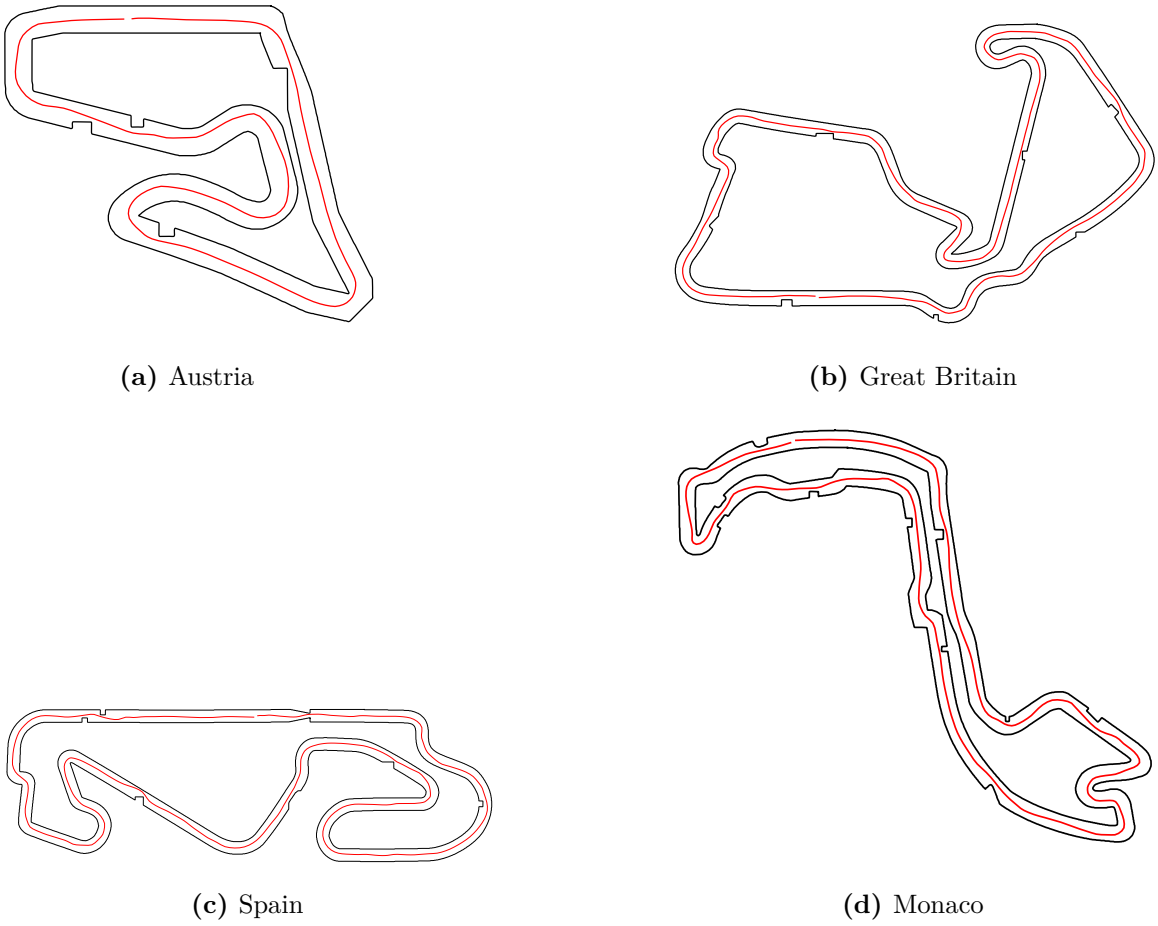
Table 9.3 shows the lap times for agents trained and tested on different combinations of the 4 baseline tracks. The results show the performance on both the seen and unseen tracks. It shows that the agent is able to generalise its racing policy to unseen tracks as it is still able to achieve competitive lap times on the unseen tracks. This shows that agent's are still racing on unseen tracks and not just cautiously navigating them.

Test	Train			
	AUT	**ESP**	**GBR**	**MCO**
AUT	15.51 ± 0.84	18.31 ± 0.83	18.39 ± 1.29	17.77 ± 1.19
ESP	38.90 ± 1.92	39.38 ± 1.13	42.57 ± 3.92	40.92 ± 2.29
GBR	34.30 ± 1.98	38.22 ± 1.58	32.24 ± 3.21	36.02 ± 2.10
MCO	29.42 ± 1.57	32.97 ± 1.16	32.31 ± 2.32	28.45 ± 1.71

Table 9.3: Lap times [s] and standard deviation of agents trained on and tested on different tracks

To further assess the agent's generalisation ability, they are tested on their ability to avoid random obstacles introduced onto the track that it is trained on. The agent has never been exposed to any inconsistencies in the track boundary in the training phase, so these features would be new to the agent. Each agent is trained on a tests tracks for 100,000 training steps with a maximum speed of 8 m/s. Next, they are tested on the modified version of the track on which they are trained.

Figure 9.2 shows the trajectories of the agents on the modified tracks. The agents are able to successfully identify obstacles and change the path to avoid them. This further shows the robustness and generalisation ability of our CO-TD3 agent.



(a) Austria

(b) Great Britain

(c) Spain

(d) Monaco

Figure 9.2: The red trajectories show the path the agents took around each of the test tracks when randomly placed obstacles are introduced along the track boundary

In summary, the results of this section highlight the advances we have made in reinforcement learning for autonomous racing, particularly in improving agents' overall performance and making them a viable alternative to classic methods. Furthermore, the generalisation ability of CO-TD3 agents enables them to operate effectively on unseen tracks, a capability that classic control approaches lack.

9.2 Physical vehicle

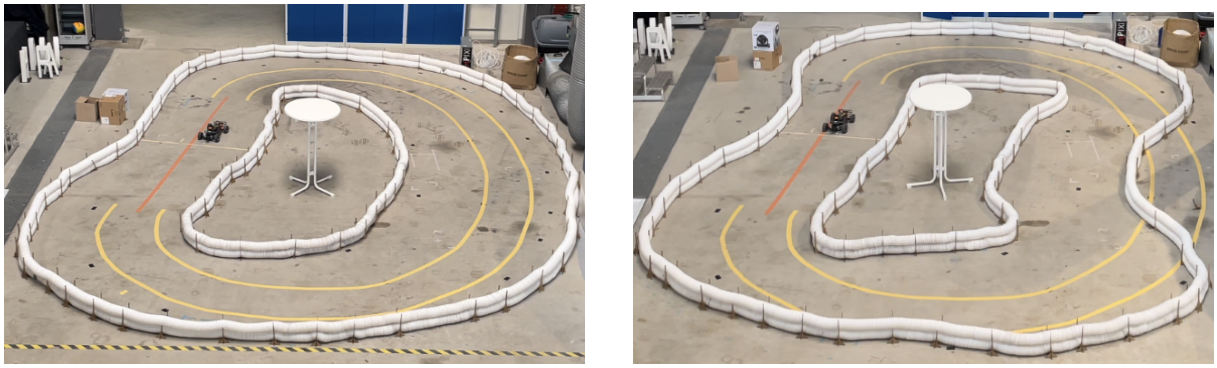
These test outline the performance of the algorithm when trained in simulation and then tested on the real vehicle. It tests how well the agent performs on a real track and compares this to the performance obtained in simulation. Furthermore, the extent to which the agent can generalise on the real track is also tested.

9.2.1 Seen tracks

Our CO-TD3 agent's ability to transfer to a real vehicle is a crucial aspect of its performance. This test shows that our agent can consistently and reliably complete laps on the actual F1TENTH vehicle on a real track. The test set-up in Figure 9.3 shows the two track variations (real track 1 and 2) where the real tests are carried out.

An agent is trained on the simulated version of real track 1 for 100,000 training steps. The performance is first tested by having it complete test laps on the simulated track before transferring it to the real vehicle. The same process is followed for a different agent for real track 2.

Our agent is able to achieve a 100% completion rate while racing for more than 130 seconds and completing more than 20 laps. The maximum speed while testing on the vehicle is limited to 3 m/s, as speeds greater than this would exceed the friction limits of the track. The track surface is made of slick concrete with very little traction. This caused lateral and longitudinal slip which prevented any higher speeds.



(a) **(b)**

Figure 9.3: Images of the test set-up with the real vehicle for both (a) real track 1 and (b) real track 2

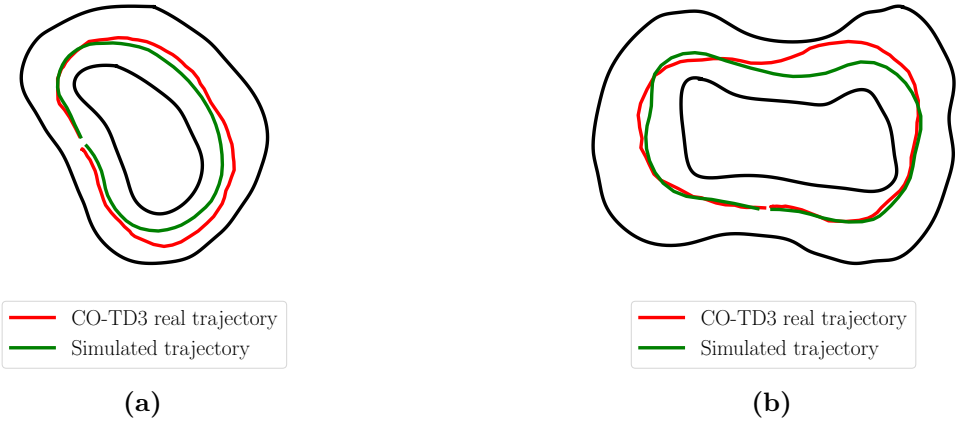
A test is performed from standstill until one lap is completed. The lap times for these tests are reported in Table 9.4. Figure 9.4 shows the track model as well as the trajectory and speed profile of both real and simulated tests on the two different physical tracks. For a direct comparison, the speed for the simulated test will also be limited to 3 m/s.

Track name	**Real lap time** [s]	**Simulated lap time** [s]
Real Track 1	6.45	6.44
Real Track 2	6.75	6.60

Table 9.4: Real and simulated lap times for the real tracks

The actual and simulated agents' performance is very similar for both of the tracks. The simulated vehicle achieved only slightly faster lap times than the real vehicle. The

difference in lap time can be the result of the wider trajectory taken by the agents on the real vehicle.



(a) **(b)**

Figure 9.4: The trajectories of the agent tested in simulation (green line) and on the real vehicle (red line) for both (a) real track 1 and (b) real track 2

Figure 9.5 and Figure 9.6 shows the speed profiles of the agents on the real track and in simulation. The real vehicle has a higher average speed, which allowed it to still achieve a competitive lap time with a less optimal trajectory. Figure 9.6 shows that the delayed response of the physical motor helped to achieve a higher average speed because it is unable to change speed at the same rate as the simulated vehicle. The major limitation of the physical system is the friction limits. The track surface limited its speed to 3 m/s, while the simulated vehicle can complete laps much faster, as the maximum speed is set to 8 m/s.

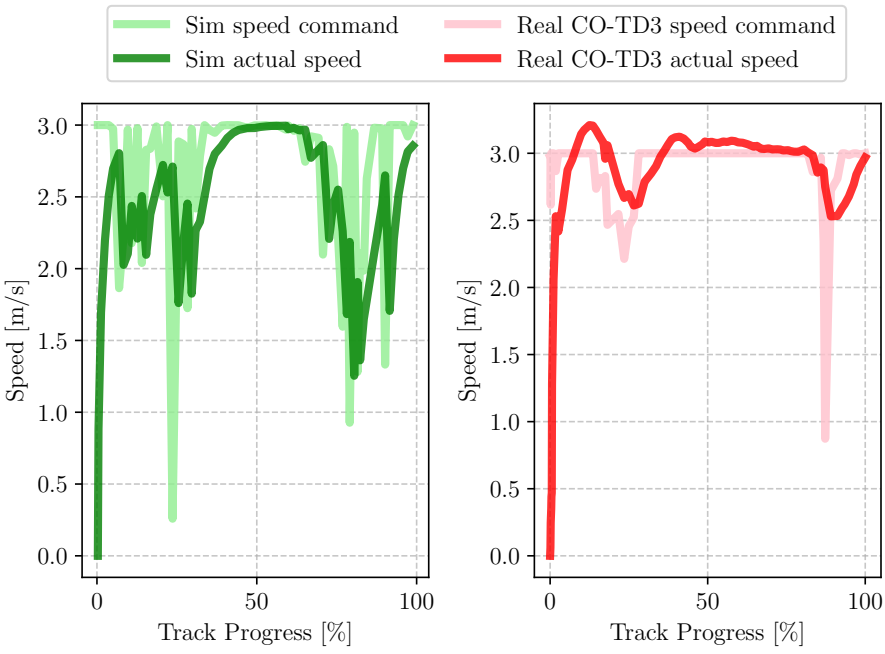


Figure 9.5: The speed profile of the real and simulated vehicle when tested on real track 1

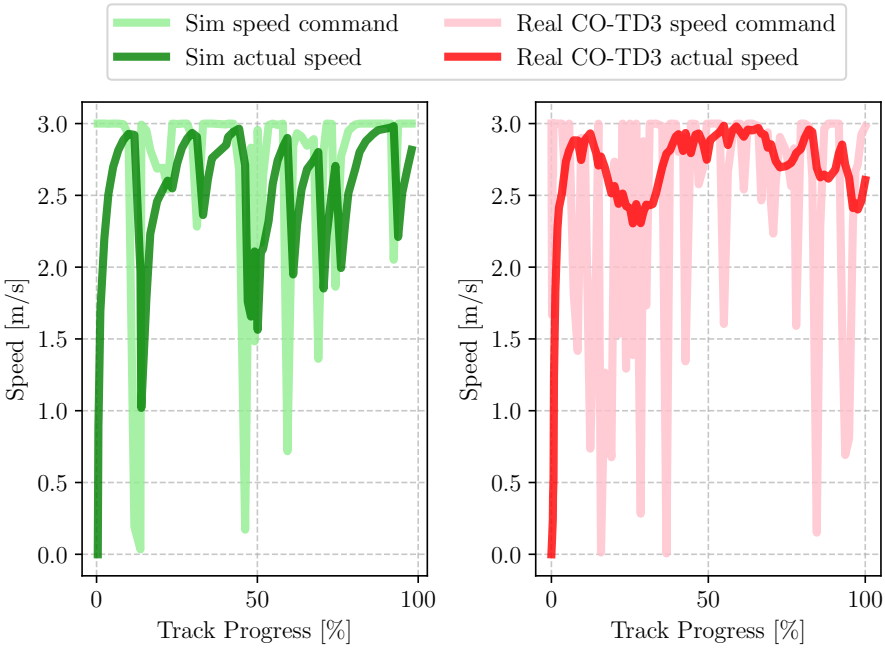
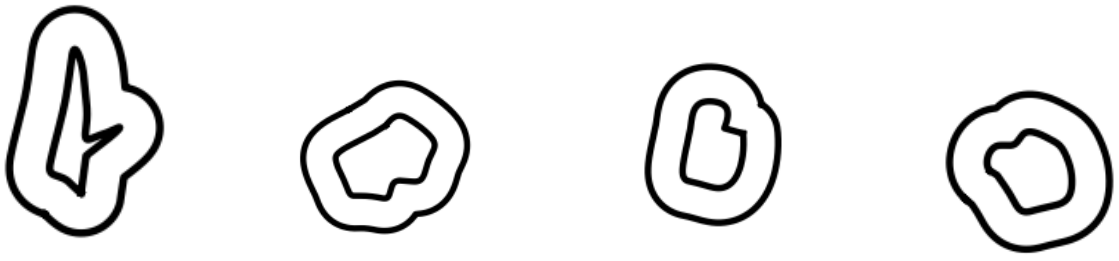


Figure 9.6: The speed profile of the real and simulated vehicle when tested on real track 2

9.2.2 Unseen tracks

The CO-TD3 agent's generalisation ability is able transfer over to the real vehicle, as it is able to perform on the real track when trained on a different track in simulation. This is tested by conducting an experiment where an agent is trained on each of the alternate tracks shown Figure 9.7, as well as the simulated version of real track 2. The agents are then transferred to the real vehicle to complete test laps on real track 1 ((a) in Figure 9.3).



(a) Alternate track 1 **(b)** Alternate track 2 **(c)** Alternate track 3 **(d)** Alternate track 4

Figure 9.7: Track outlines

Table 9.5 shows the performance of these agents compared to an agent trained on the simulated version of the real track 1 to act as a seen track baseline. This experiment shows that the agents are able to race without crashing and can achieve lap times very similar to the seen track agent when performing on unseen tracks on the physical vehicle. This shows that the agent's generalisation ability is able to transfer over to the real vehicle.

Track Trained on	Lap Time [s]
Real track 1	6.45
Real track 2	6.45
Alternate 1	6.27
Alternate 2	6.47
Alternate 3	6.60
Alternate 4	7.10

Table 9.5: Mean lap times [s] for agents trained on various tracks and tested on real track 1

Much like with the simulated tests, the generalisation ability is further tested by introducing random obstacles onto the track. Once again, the agent that is tested on the simulated version of real track 1 has never been exposed to any kind of obstacle during training, so these are unknown to the agent. Figure 9.8 shows that the agent is able to successfully adapt to obstacles on the track. This shows how well the agent can generalise to a set of new features that are different from anything seen in training.

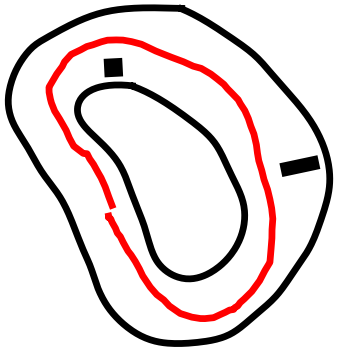


Figure 9.8: The trajectory of an agent on real track 1 with added obstacles

In summary, physical vehicle experiments demonstrate the effectiveness of the CO-TD3 agent in transitioning from simulated to real world environments. The agent's ability to achieve a 100% completion rate on real seen tracks showcases its reliable performance when tested on actual F1TENTH vehicles. Additionally, the successful completion of laps on unseen tracks highlights the agent's robust generalisation capabilities, reinforcing the potential of reinforcement learning techniques in autonomous racing. Achieving these hardware results is particularly significant as much of the existing end-to-end work is performed in simulated environments and gaming contexts. These results indicate that the CO-TD3 agent not only bridges the gap between simulation and reality, but also shows the viability of DRL methods in autonomous racing.

Chapter 10

Conclusion

Autonomous racing on unseen tracks is a challenging task requiring algorithms to have the ability to generalise while still maintaining competitive racing performance. We evaluate the use of a twin delayed deep deterministic policy gradient (TD3) algorithm for the task of racing on unseen tracks in both a simulated and a real environment according to the objectives outlined in Chapter 1. We also highlight the capabilities of this approach and how it differs from current high-performing autonomous racing techniques. Finally, potential future work building on this research is discussed.

10.1 Evaluation of autonomous racing with CO-TD3

The objectives of this research were to demonstrate that deep reinforcement learning (DRL) is a viable approach for autonomous racing. This involves enhancing the performance of end-to-end autonomous racing methods to have performance comparable to classic control algorithms. To do this, many limitations pertaining to current end-to-end methods needed to be addressed. The most detrimental limitation of current end-to-end methods was inconstant action selection, which often resulted in inefficient racing trajectories and slower lap time compared to classic algorithms. Furthermore, the reliance of these algorithms on simulated environments limited the amount of real-world data obtained to validate their use on physical vehicles. Addressing these limitations would not only enable competitive racing on seen tracks, but also extend their use to unseen tracks in both simulation and reality, which is an ability classic control methods do not possess.

To address these limitations, we created a centre-orientated TD3 agent (CO-TD3) that uses a real-time centring term to better position itself on the track. This, coupled with an agent state vector consisting of a LiDAR scan and a measurement of its current actions, allowed our agent to achieve more consistence action selection. We then developed a comprehensive reward function that encourages the agent to maximise its reward by completing safe and fast laps. In Chapter 9, we evaluated the performance of our CO-TD3 agents by setting up experiments to compare the performance to a benchmark of other racing algorithms on the F1TENTH platform. This showed that our CO-TD3 agents were able to compete with classical control algorithms and outperform previous end-to-end

methods. Furthermore, our agents' ability to generalise is shown by using the random track generator developed in Chapter 7, to create a set of unseen tracks. This showed that our agent can achieve consistent performance on these tracks, which is the main differentiable feature between our method and pre-existing autonomous racing algorithms.

Using the ROS framework designed in Chapter 8, we were able to transfer our CO-TD3 agent to a real vehicle and perform tests on a real track. This shows the robustness of our algorithm, as it was able to perform seamlessly in the real world without any additional training adjustments.

The results achieved show that our CO-TD3 approach achieves the objectives outlined in Section 1.2. However, this method does have limitations, as uncertainty of what makes a track ideal for generalisation means that agents trained on a certain track will not always perform as intended. Agents have to be trained on multiple tracks and thoroughly tested to ensure that a track produces agents with the required general performance.

The agent's ability to reach this level of performance using only learnt behaviour from sensor measurements demonstrates the remarkable potential of DRL methods. The capacity to extract relationships between inputs and outputs, and to generalise this understanding to unseen tracks in both simulated and real-world environments is a true testament of this algorithms capabilities.

10.2 Future work

In this thesis, we demonstrated that a TD3 racing agent can effectively navigate unseen obstacles and adapt to unseen racetracks, showcasing the superior generalisation capabilities of DRL-based agents over classical methods. This adaptability opens avenues for expanding autonomous racing into more dynamic scenarios, such as competitive head-to-head racing, where agents must make complex decisions based on the constantly evolving positions and actions of other vehicles.

The intricacy of competitive racing, where the actions of each vehicle directly influence the other's trajectory, restricts the feasibility of traditional control algorithms. Instead, a DRL-based solution, which leverages the generalisation abilities demonstrated in unseen environments, presents a promising approach for head-to-head racing. Further development of this technology could enable agents to dynamically optimise their racing strategies, thereby addressing the complex and interactive decision-making required in competitive autonomous racing.

Moreover, the success of DRL in autonomous racing serves as a step toward deploying algorithms like these in real-world driving scenarios. As DRL agents continue to advance in handling complex and unpredictable environments, their application could be extended beyond racing to broader autonomous driving challenges, such as navigating mixed traffic urban settings. These settings will expose autonomous vehicles to diverse dynamic

situations in which DRL agents would have to interact with and predict other drivers' motion and react accordingly while obeying traffic rules. This is a challenging task that relies on autonomous algorithms to react according to real-time data and not rely on predefined plans. The generalisation ability of DRL agents can hopefully enhance this ability and increase the safety of passenger vehicles.

Bibliography

[1] "Abu Dhabi Autonomous Racing League in UAE — A2RL — a2rl.io," https://a2rl.io/, [Accessed 02-11-2024].

[2] "Indy Autonomous Challenge — indyautonomouschallenge.com," https://www.indyautonomouschallenge.com/, [Accessed 02-11-2024].

[3] "F1TENTH — f1tenth.org," https://f1tenth.org/build.html, [Accessed 23-05-2024].

[4] M. Brown and J. C. Gerdes, "Coordinating tire forces to avoid obstacles using nonlinear model predictive control," *IEEE Transactions on Intelligent Vehicles*, vol. 5, pp. 21–31, 3 2020.

[5] V. Sukhil and M. Behl, "Adaptive lookahead pure-pursuit for autonomous racing," 11 2021. [Online]. Available: http://arxiv.org/abs/2111.08873

[6] J. Betz, H. Zheng, A. Liniger, U. Rosolia, P. Karle, M. Behl, V. Krovi, and R. Mangharam, "Autonomous vehicles on the edge: A survey on autonomous vehicle racing," *IEEE Open Journal of Intelligent Transportation Systems*, vol. 3, pp. 458–488, 2022.

[7] R. S. Sutton and A. G. Barto, *Reinforcement learning : An introduction*.

[8] M. Laskin, D. Yarats, H. Liu, K. Lee, A. Zhan, K. Lu, C. Cang, L. Pinto, and P. Abbeel, "URLB: Unsupervised Reinforcement Learning Benchmark," *CoRR*, vol. abs/2110.15191, 2021. [Online]. Available: https://arxiv.org/abs/2110.15191

[9] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, "Quantifying generalization in reinforcement learning," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 1282–1289. [Online]. Available: https://proceedings.mlr.press/v97/cobbe19a.html

[10] S. D. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. H. Eng, D. Rus, and M. H. Ang, "Perception, planning, control, and coordination for autonomous vehicles," *Machines*, vol. 5, no. 1, 2017. [Online]. Available: https://www.mdpi.com/2075-1702/5/1/6

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

[12] B. Paden, M. Cap, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," 2016.

[13] M. Schwenzer, M. Ay, T. Bergs, and D. Abel, "Review on model predictive control: An engineering perspective," *The International Journal of Advanced Manufacturing Technology*, vol. 117, pp. 1–23, 11 2021.

[14] E. Alcalá, V. Puig, J. Quevedo, and U. Rosolia, "Autonomous racing using Linear Parameter Varying-Model Predictive Control (LPV-MPC)," *Control Engineering Practice*, vol. 95, p. 104270, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0967066119302187

[15] M. Althoff, M. Koschi, and S. Manzinger, "Commonroad: Composable benchmarks for motion planning on roads," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017, pp. 719–726.

[16] Y. Yue, Z. Wang, J. Liu, and G. Li, "Multi-scenario Learning MPC for Automated Driving in Unknown and Changing Environments," in *2023 IEEE 21st International Conference on Industrial Informatics (INDIN)*, 2023, pp. 1–6.

[17] N. R. Kapania and J. C. Gerdes, "Design of a feedback-feedforward steering controller for accurate path tracking and stability at the limits of handling," *Vehicle System Dynamics*, vol. 53, no. 12, pp. 1687–1704, 2015. [Online]. Available: https://doi.org/10.1080/00423114.2015.1055279

[18] J. Becker, N. Imholz, L. Schwarzenbach, E. Ghignone, N. Baumann, and M. Magno, "Model- and acceleration-based pursuit controller for high-performance autonomous racing," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 5276–5283.

[19] H. B. Pacejka and E. Bakker, "The magic formula tyre model," *Vehicle System Dynamics*, vol. 21, no. sup001, pp. 1–18, 1992. [Online]. Available: https://doi.org/10.1080/00423119208969994

[20] V. Sezer and M. Gokasan, "A novel obstacle avoidance algorithm: "follow the gap method"," *Robotics and Autonomous Systems*, vol. 60, no. 9, pp. 1123–1134, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889012000838

[21] M. D. Demir and V. Sezer, "Autonomous overtaking maneuver design based on follow the gap method." in *ICINCO (1)*, 2019, pp. 295–302.

[22] T. Hossain, H. Habibullah, R. Islam, and R. V. Padilla, "Local path planning for autonomous mobile robots by integrating modified dynamic-window approach and

improved follow the gap method," *Journal of Field Robotics*, vol. 39, no. 4, pp. 371–386, 2022.

[23] R. Amsters and P. Slaets, "Turtlebot 3 as a robotics education platform," in *Robotics in Education*, M. Merdan, W. Lepuschitz, G. Koppensteiner, R. Balogh, and D. Obdržálek, Eds. Cham: Springer International Publishing, 2020, pp. 170–181.

[24] T. Johannink, S. Bahl, A. Nair, J. Luo, A. Kumar, M. Loskyll, J. A. Ojea, E. Solowjow, and S. Levine, "Residual reinforcement learning for robot control," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 6023–6029.

[25] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine, "How to train your robot with deep reinforcement learning: Lessons we have learned," *The International Journal of Robotics Research*, vol. 40, no. 4-5, pp. 698–721, 2021. [Online]. Available: https://doi.org/10.1177/0278364920987859

[26] H. Zhu, J. Yu, A. Gupta, D. Shah, K. Hartikainen, A. Singh, V. Kumar, and S. Levine, "The ingredients of real-world robotic reinforcement learning," 2020.

[27] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[28] P. Cai, X. Mei, L. Tai, Y. Sun, and M. Liu, "High-speed autonomous drifting with deep reinforcement learning," 1 2020. [Online]. Available: http://arxiv.org/abs/2001.01377http://dx.doi.org/10.1109/LRA.2020.2967299

[29] M. Jaritz, R. de Charette, M. Toromanoff, E. Perot, and F. Nashashibi, "End-to-end race driving with deep reinforcement learning," 7 2018. [Online]. Available: http://arxiv.org/abs/1807.02371

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

[31] F. Fuchs, Y. Song, E. Kaufmann, D. Scaramuzza, and P. Dürr, "Super-human performance in gran turismo sport using deep reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 4257–4264, 2021.

[32] K. Stachowicz, D. Shah, A. Bhorkar, I. Kostrikov, and S. Levine, "Fastrlap: A system for learning high-speed driving via deep rl and autonomous practicing," 2023. [Online]. Available: https://sites.google.com/view/fastrlap.

[33] M. Bosello, R. Tse, and G. Pau, "Train in austria, race in montecarlo: Generalized rl for cross-track f1tenthlidar-based races." Institute of Electrical and Electronics Engineers Inc., 2022, pp. 290–298.

[34] A. Brunnbauer, L. Berducci, A. Brandstätter, M. Lechner, R. Hasani, D. Rus, and R. Grosu, "Latent imagination facilitates zero-shot transfer in autonomous racing," 3 2021. [Online]. Available: http://arxiv.org/abs/2103.04909

[35] R. Ivanov, T. J. Carpenter, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Case study: Verifying the safety of an autonomous racing car with a neural network controller," 10 2019. [Online]. Available: http://arxiv.org/abs/1910.11309

[36] B. Evans, H. Engelbrecht, and H. Jordaan, "High-speed autonomous racing using trajectory-aided deep reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 8, pp. 5353–5359, 09 2023.

[37] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation learning: A survey of learning methods," *ACM Comput. Surv.*, vol. 50, no. 2, apr 2017. [Online]. Available: https://doi.org/10.1145/3054912

[38] Y. Pan, C.-A. Cheng, K. Saigol, K. Lee, X. Yan, E. Theodorou, and B. Boots, "Agile autonomous driving using end-to-end deep imitation learning," 9 2017. [Online]. Available: http://arxiv.org/abs/1709.07174

[39] J. Zhang and K. Cho, "Query-efficient imitation learning for end-to-end simulated driving," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, Feb. 2017. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/10857

[40] M. Brunner, U. Rosolia, J. Gonzales, and F. Borrelli, "Repetitive learning model predictive control: An autonomous racing example," 12 2017, pp. 2545–2550.

[41] U. Rosolia and F. Borrelli, "Learning how to autonomously race a car: A predictive control approach," 2019. [Online]. Available: https://arxiv.org/abs/1901.08184

[42] C. Vallon and F. Borrelli, "Task decomposition for iterative learning model predictive control," 2020. [Online]. Available: https://arxiv.org/abs/1903.07003

[43] E. Ghignone, N. Baumann, and M. Magno, "TC-Driver: A Trajectory Conditioned Reinforcement Learning Approach to Zero-Shot Autonomous Racing," *Field Robotics*, vol. 3, no. 1, p. 637–651, Jan. 2023. [Online]. Available: http://dx.doi.org/10.55417/fr.2023020

[44] K. L. R. Talvala and J. C. Gerdes, "Lanekeeping at the limits of handling: Stability via lyapunov functions and a comparison with stability control," 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:111531907

[45] K. Kritayakirana and J. C. Gerdes, "Autonomous vehicle control at the limits of handling," *International Journal of Vehicle Autonomous Systems*, vol. 10, p. 271, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:108898921

[46] H.-K. Park and J. C. Gerdes, "Optimal tire force allocation for trajectory tracking with an over-actuated vehicle," *2015 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1032–1037, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:1050201

[47] J. Ni and J. Hu, "Path following control for autonomous formula racecar: Autonomous formula student competition," in *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017, pp. 1835–1840.

[48] J. Ni, J. Hu, and C. Xiang, "A review for design and dynamics control of unmanned ground vehicle," *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, vol. 235, no. 4, pp. 1084–1100, 2021. [Online]. Available: https://doi.org/10.1177/0954407020912097

[49] E. Wachter, A. Schmeitz, F. Bruzelius, and M. Alirezaei, *Path Control in Limits of Vehicle Handling: A Sensitivity Analysis*, 02 2020, pp. 1089–1095.

[50] C. E. Beal and J. C. Gerdes, "Model predictive control for vehicle stabilization at the limits of handling," *IEEE Transactions on Control Systems Technology*, vol. 21, pp. 1258–1269, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:27738444

[51] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, "Aggressive driving with model predictive path integral control," *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1433–1440, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:18322548

[52] R. Verschueren, M. Zanon, R. Quirynen, and M. Diehl, "Time-optimal race car driving using an online exact hessian based nonlinear MPC algorithm," in *2016 European Control Conference (ECC)*, 2016, pp. 141–147.

[53] B. Alrifaee and J. Maczijewski, "Real-time trajectory optimization for autonomous vehicle racing using sequential linearization," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, 2018, pp. 476–483.

[54] P. Scheffe, T. M. Henneken, M. Kloock, and B. Alrifaee, "Sequential convex programming methods for real-time optimal trajectory planning in autonomous vehicle racing," *IEEE Transactions on Intelligent Vehicles*, vol. 8, no. 1, pp. 661–672, 2023.

[55] T. Novi, A. Liniger, R. Capitani, and C. Annicchiarico, "Real-time control for at-limit handling driving on a predefined path," *Vehicle System Dynamics*, vol. 58, pp. 1007 – 1036, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:146057554

[56] Y. Liu, L. Shi, W. Xu, X. Xiong, W. Sun, and L. Qu, "Design of Driverless Racing Chassis Based on MPC," 11 2020, pp. 6061–6066.

[57] M. S. Gandhi, B. Vlahov, J. Gibson, G. Williams, and E. A. Theodorou, "Robust model predictive path integral control: Analysis and performance guarantees," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, p. 1423–1430, Apr. 2021. [Online]. Available: http://dx.doi.org/10.1109/LRA.2021.3057563

[58] F. K. Pour, D. Theilliol, V. Puig, and G. Cembraño, "Health-aware control design based on remaining useful life estimation for autonomous racing vehicle." *ISA transactions*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:218894812

[59] N. Li, É. Goubault, L. Pautet, and S. Putot, "Autonomous racecar control in head-to-head competition using mixed-integer quadratic programming." [Online]. Available: https://api.semanticscholar.org/CorpusID:268032600

[60] E. Perot, M. Jaritz, M. Toromanoff, and R. De Charette, "End-to-end driving in a realistic racing game with deep reinforcement learning," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2017, pp. 474–475.

[61] T. de Bruin, J. Kober, K. Tuyls, and R. Babuška, "Integrating state representation learning into deep reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1394–1401, 2018.

[62] E. Chisari, A. Liniger, A. Rupenyan, L. V. Gool, and J. Lygeros, "Learning from simulation, racing in reality," 2021. [Online]. Available: https://arxiv.org/abs/2011.13332

[63] L. Hewing, A. Liniger, and M. N. Zeilinger, "Cautious NMPC with Gaussian Process Dynamics for Autonomous Miniature Race Cars," in *2018 European Control Conference (ECC)*. IEEE, Jun. 2018. [Online]. Available: http://dx.doi.org/10.23919/ECC.2018.8550162

[64] A. Jain, M. O'Kelly, P. Chaudhari, and M. Morari, "Bayesrace: Learning to race autonomously using prior experience," 2020. [Online]. Available: https://arxiv.org/abs/2005.04755

[65] C. Szepesvári, *Algorithms for Reinforcement Learning*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.

[66] H. Dong, Z. Ding, and S. Zhang, *Deep Reinforcement Learning Fundamentals, Research and Applications: Fundamentals, Research and Applications*, 01 2020.

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

[68] B. Singh, R. Kumar, and V. Singh, "Reinforcement learning in robotic applications: a comprehensive survey," *Artificial Intelligence Review*, vol. 55, 02 2022.

[69] S. Dankwa and W. Zheng, "Twin-Delayed DDPG: A Deep Reinforcement Learning Technique to Model a Continuous Movement of an Intelligent Robot Agent." Association for Computing Machinery, 8 2019.

[70] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2018. [Online]. Available: https://arxiv.org/abs/1802.09477

[71] B. D. Evans, R. Trumpp, M. Caccamo, F. Jahncke, J. Betz, H. W. Jordaan, and H. A. Engelbrecht, "Unifying f1tenth autonomous racing: Survey, methods and benchmarks," 2024.

[72] F. Fuchs, Y. Song, E. Kaufmann, D. Scaramuzza, and P. Durr, "Super-human performance in gran turismo sport using deep reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 6, pp. 4257–4264, 7 2021.

[73] B. D. Evans, H. W. Jordaan, and H. A. Engelbrecht, "Comparing deep reinforcement learning architectures for autonomous racing," *Machine Learning with Applications*, vol. 14, p. 100496, 12 2023.

[74] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[75] C. Zhang, O. Vinyals, R. Munos, and S. Bengio, "A study on overfitting in deep reinforcement learning," 2018. [Online]. Available: https://arxiv.org/abs/1804.06893

[76] E. Valassakis, Z. Ding, and E. Johns, "Crossing the gap: A deep dive into zero-shot sim-to-real transfer for dynamics," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 5372–5379.

[77] S. Macenski and I. Jambrecic, "SLAM Toolbox: SLAM for the dynamic world," *Journal of Open Source Software*, vol. 6, no. 61, p. 2783, 2021. [Online]. Available: https://doi.org/10.21105/joss.02783

[78] C. Walsh and S. Karaman, "CDDT: Fast Approximate 2D Ray Casting for Accelerated Localization," vol. abs/1705.01167, 2017. [Online]. Available: http://arxiv.org/abs/1705.01167

[79] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 9 2015. [Online]. Available: http://arxiv.org/abs/1509.02971

[80] S. Zhang and R. S. Sutton, "A deeper look at experience replay," 2018. [Online]. Available: https://arxiv.org/abs/1712.01275

[81] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," 2019. [Online]. Available: https://arxiv.org/abs/1709.06560

Appendix A

Hyperparameter tuning

A.1 TD3 hyperparameters

The hyperparameter tuning for the TD3 algorithm is investigated here. The starting values are shown in Table A.1, and when altering the specific hyperparameters, the other hyperparameters are kept to these original values.

Parameter	Value	Description
Action noise	0.2	Noise added to actions
Batch size	100	Size of the batch
Discount factor	0.99	Discount factor gamma
Learning rate	1e-3	Learning rate of the agent
Policy update frequency	4	Number of iterations to wait before updating the policy network

Table A.1: Initial parameter settings for the model

A.1.1 Action noise

The action noise is added to the action after the actor target network generates an action based on the current state (Equation 3.15). Increasing action noise can help increase exploration and improve robustness and generalisation, however, if it is too large, the agent will struggle to find an optimal policy, leading to instability in learning outcomes [70, 79].

Figure A.1 shows the effect of varying the action noise. It is clear that the higher noise allowed the agent to learn faster, but it struggled to converge. The other values showed slower convergence but more stability. A noise value of 0.2 showed a good balance between the training speed and convergence with a small standard deviation toward the end of training.

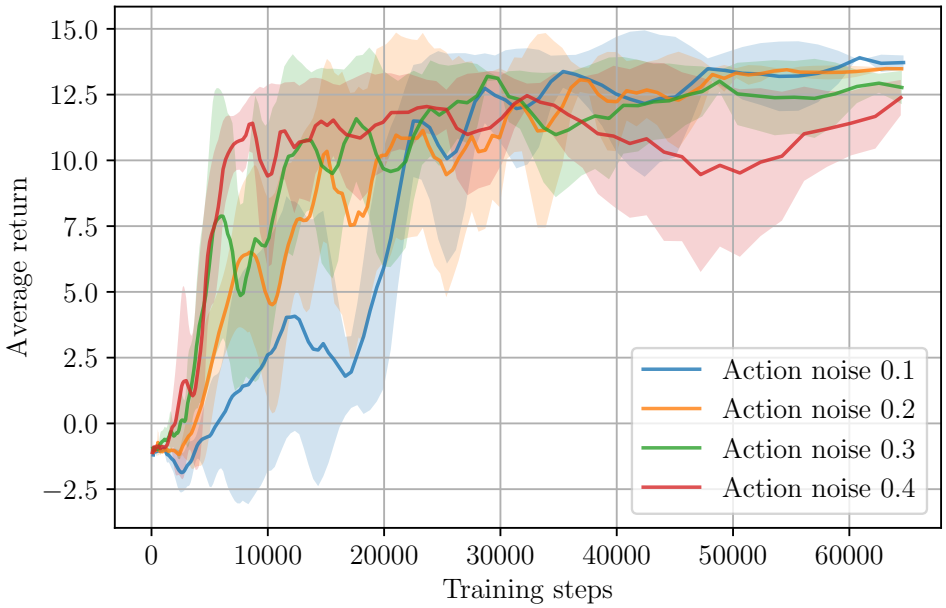


Figure A.1: The average return and standard deviation (shaded) of agents with varying action noise values

A.1.2 Policy update frequency

Delaying the policy update is done to stabilise the agent's training by updating the critics Q_θ more frequently than the actor π_ϕ (line 13 in Algorithm 3.1). Since the actor relies on critic, more accurate Q-values lead to a better actor. If the actor is updated more infrequently, the critic has more time to learn the rewards and transitions of the environment, resulting in more reliable policy updates. However, lowering the frequency too much causes a decrease in the learning rate as the actor is not being updated enough to learn optimally [70].

Figure A.2 shows that TD3 is quite robust to changes in the frequency of the policy update. A frequency of 2 shows more instability than the other values; therefore, updating the policy less at a frequency of 8 would be more beneficial to training stability. This means that for every 8 critic updates, the actor is updated once.

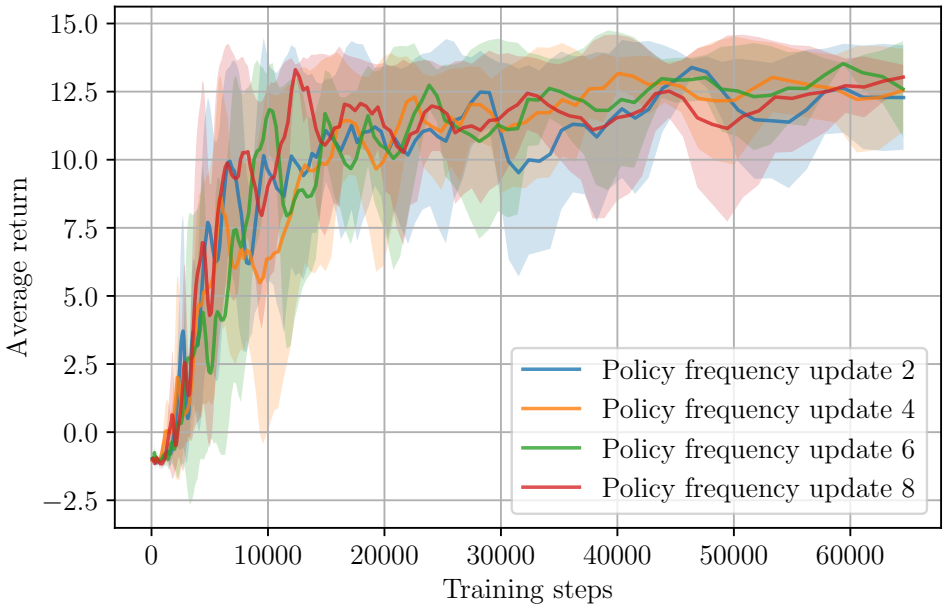


Figure A.2: The average return and standard deviation (shaded) of agents with varying policy update frequencies

A.1.3 Discount factor

The discount factor controls how much emphasis is placed on short-term versus long-term reward. A low discount factor prioritises immediate rewards, causing the agent to focus on short-term gains while paying less attention to future outcomes. This can result in greedy behaviour, where the agent makes decisions that have immediate benefits to the detriment of overall performance. Conversely, a high discount factor encourages the agent to plan for the long term by valuing future rewards more heavily. This can lead to strategic actions that optimise cumulative rewards over time, but it can slow the learning process, especially in environments like ours with delayed rewards [80]. The discount factor is used when computing the target Q-value form the critic networks (Equation 3.16)

Figure A.3 shows that higher discount values led to a higher overall average reward as the agent learnt to be more strategic to ensure the highest completion reward. Therefore, 0.99 will be used.

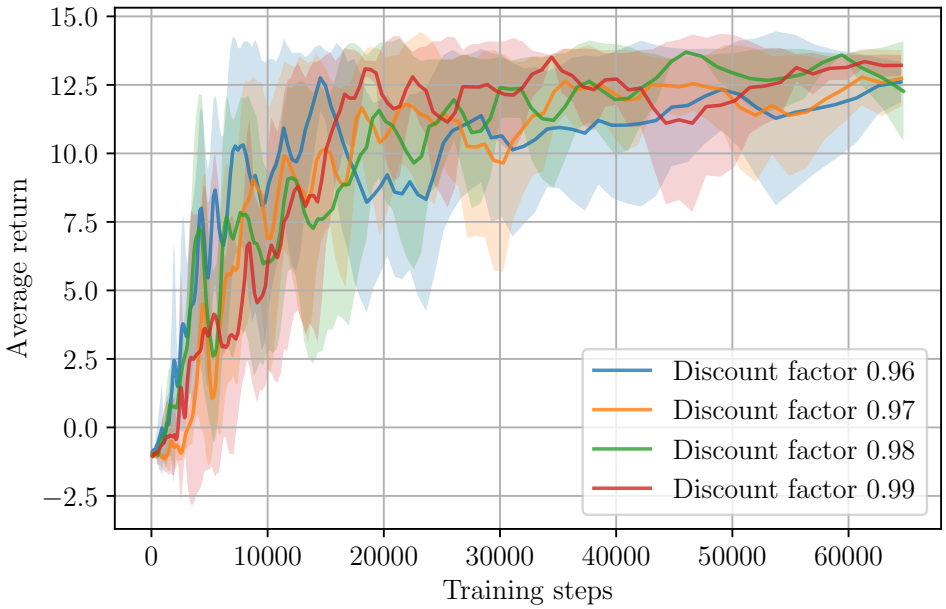


Figure A.3: The average return and standard deviation (shaded) of agents with varying discount factors

A.1.4 Learning rate

The learning rate refers to the speed with which the agent learns, since it is the size of the steps that the agent takes to update the parameters during training (Equation 3.10, Equation 3.13). The agent will take longer to find an optimal policy if it is too small, resulting in longer training times and increased computational costs. Additionally, this can cause the agent to get stuck in sub-optimal local minima, slowing progress. Conversely, suppose that the learning rate is too large. In that case, the agent can overshoot the optimal parameter values of the policy, leading to an oscillation in training and a failure to learn consistently [81].

The learning rate had a large impact on the agent's ability to learn. Figure A.4 shows that large learning rates struggle to learn consistently and produce unstable behaviour. At very large values, the agent is incapable of learning at all, as the update steps are too big. The ideal learning rate here is $1e-4$, which is an order of magnitude smaller than the initial value. This value showed smooth and stable learning without increasing the learning time.

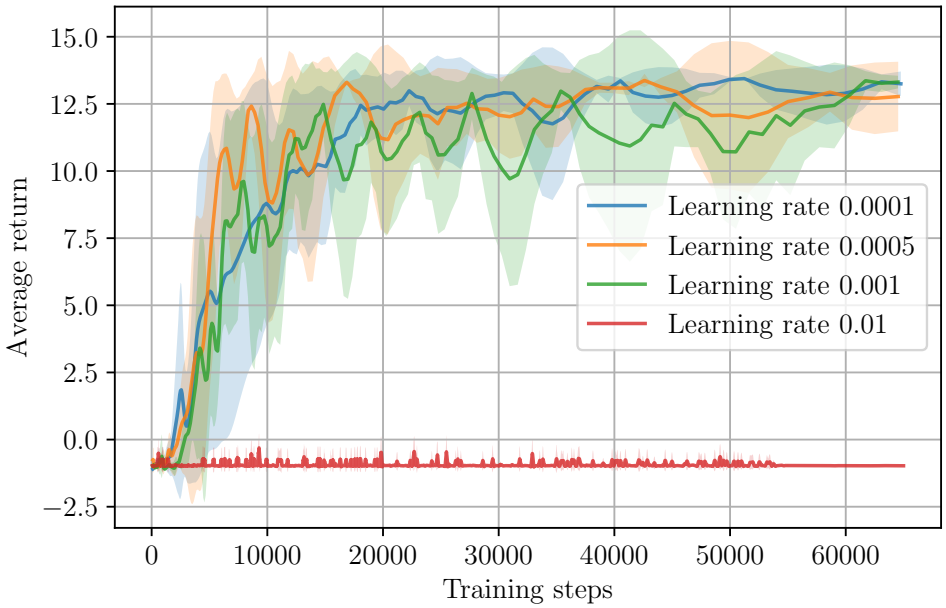


Figure A.4: The average training curve of agents with varying learning rates

A.1.5 Batch size

The batch size is the number of samples drawn from the replay buffer in each training step. Small batch sizes may lead to noisier updates and less stability, as the samples are coming from a smaller batch with less variance in the samples. Large batch sizes increase the diversity of samples, resulting in more stable training. Although large batches stabilise learning, they can hinder the agent's ability to quickly adapt to new environments, which can be important in dynamic tasks such as racing.

Varying the batch size had quite a small effect on overall training as seen in Figure A.5. However, there was a noticeable increase in training time as the batch size grew. As the smallest batch size results in good training performance, a batch size of 64 will be used.

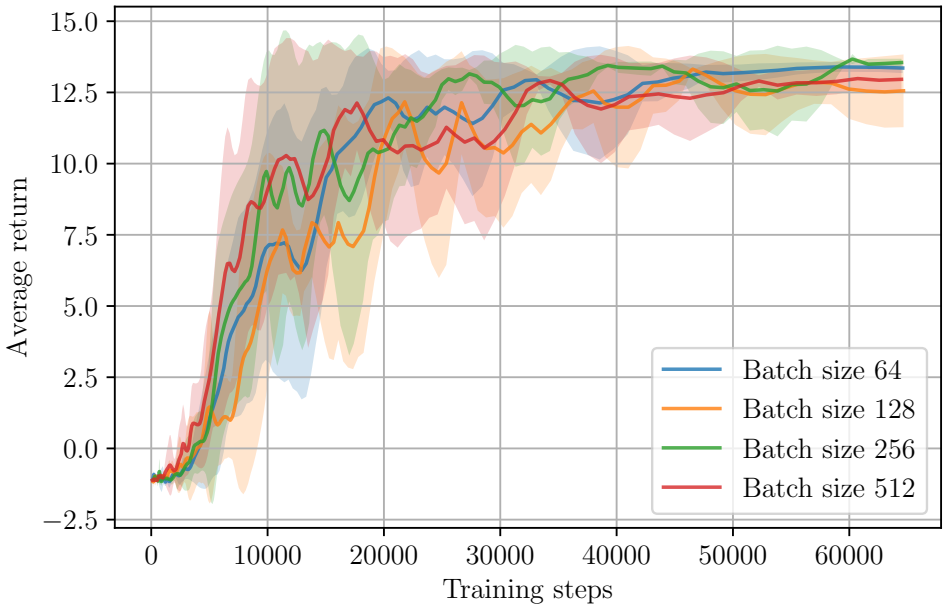


Figure A.5: The average return and standard deviation (shaded) of agents with varying batch sizes

The final hyperparameters used are shown with all the other network parameters in Table A.2.

Parameter	Value
Action noise	0.2
Batch size	64
Discount factor	0.99
Learning rate	1e-4
Policy update frequency	8
Hidden layers	2
Hidden layer size	200
Input layer size	31
Output layer size	2
Noise clip	0.5
Polyak update weight	0.005

Table A.2: Final parameter settings for the model

A.2 Optimality assessment

To ensure that we can produce the most optimal agents, an assessment of current hyper-parameters and the composition of the state vector was performed. This shows that any

deviation from the current state of the algorithm resulted in poorer performance.

A.2.1 Hyperparameter optimality

To ensure that the chosen hyperparameters are indeed the optimal combination, an evaluation was performed to identify the effect that altering one of the hyperparameters has on training. Table A.3 shows the results of this assessment. The top row shows the optimal combination identified in Section 5.4. Each row shows a combination of hyperparameters, with one hyperparameter varied from the optimal baseline (highlighted in bold). The normalised average return from training is reported, with values normalised relative to the optimal combination. The table shows that any deviation from any of the optimal hyperparameters causes a decrease in the average return, indicating that the hyperparameters used are the most optimal.

Learning rate	**Frequency update**	**Discount factor**	**Batch size**	**Action noise**	**Normalised mean return**
$1e^{-4}$	8	0.99	64	0.2	1.000
$\mathbf{5e^{-4}}$	8	0.99	64	0.2	0.914
$\mathbf{1e^{-3}}$	8	0.99	64	0.2	0.872
$1e^{-4}$	**6**	0.99	64	0.2	0.879
$1e^{-4}$	**10**	0.99	64	0.2	0.873
$1e^{-4}$	8	**0.98**	64	0.2	0.763
$1e^{-4}$	8	0.99	**32**	0.2	0.972
$1e^{-4}$	8	0.99	**128**	0.2	0.884
$1e^{-4}$	8	0.99	64	**0.1**	0.590
$1e^{-4}$	8	0.99	64	**0.3**	0.791

Table A.3: Normalised mean return for various hyperparameter

A.2.2 Reward weights assessment

To ensure that the weight parameters for the reward function are still optimal after tuning the network hyperparameter, a subset of different weights were tested to record the same information as in Figure 5.5. It shows that the initial combination of reward function weights, highlighted by the lines, still produces the highest average normalised return when testing. Therefore, they will be used as the final value.

Centring penalty w_2	**Completion bonus w_3**	**Normalised return**
4	0.05	0.905189
4	0.10	0.932676
4	0.15	0.847668
5	0.05	0.954225
5	0.10	0.899812
5	0.15	0.859362
6	0.05	0.934431
6	0.10	0.894573
6	0.15	0.858301

Table A.4: Normalised return for different reward function weights