Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

End-to-end control on F1/10 Autonomous Car using Neural Network

Marek Žuffa

Supervisor: Ing. Jaroslav Klapálek
Field of study: Artificial intelligence
May 2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Žuffa Marek**	Personal ID number:	**483799**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence**		

II. Master’s thesis details

Master’s thesis title in English:

End-to-end control on F1/10 Autonomous Car using Neural Network

Master’s thesis title in Czech:

End-to-end řízení F1/10 autonomního auta s využitím neuronových sítí

Guidelines:

1. Get familiar with ROS2 and F1/10 autonomous racing cars.
2. Conduct a review of using reinforcement learning for end-to-end control on cars. Consider both model-free and model-based approaches.
3. Select and implement at least one model-free and at least one model-based reinforcement learning method for neural networks.
4. Tune the hyper-parameters of both networks in the simulation, compare their performance.
5. Using the conclusions from the simulator, deploy and test both networks on the F1/10 car. Identify and discuss the simulation-to-reality gap.
6. Compare the performance of implemented solutions with reactive control approaches (e.g., Follow the Gap algorithm).
7. Evaluate your results and document everything thoroughly.

Bibliography / sources:

1. S. Macenski et al., "Robot Operating System 2: Design, architecture, and uses in the wild," Science Robotics vol. 7, May 2022, doi: 10.1126/scirobotics.abm6074
2. J. Betz et al., "Autonomous Vehicles on the Edge: A Survey on Autonomous Vehicle Racing," in IEEE Open Journal of Intelligent Transportation Systems, vol. 3, pp. 458-488, 2022, doi: 10.1109/OJITS.2022.3181510
3. A. Brunnbauer et al., "Latent Imagination Facilitates Zero-Shot Transfer in Autonomous Racing," 2022 International Conference on Robotics and Automation (ICRA), Philadelphia, PA, USA, 2022, pp. 7513-7520, doi: 10.1109/ICRA46639.2022.9811650

Name and workplace of master's thesis supervisor:

Ing. Jaroslav Klapálek   Department of Control Engineering FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **18.09.2023**   Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **16.02.2025**

Ing. Jaroslav Klapálek Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to express my gratitude to Ing. Jaroslav Klapálek for the valuable information, guidance, and help while writing this thesis. Furhtermore, I would also like to thank my family and friends for all their support during the writing of this thesis, as well as throughout my studies.

Declaration

Abstract

This thesis focuses on autonomous end-to-end control of an F1TENTH model car in a single-agent racing environment. After reviewing already existing solutions, two methods were selected: a model-free algorithm, TD3, and a model-based algorithm, Dreamer. The environments for simulation and real-life scenarios are described. The methods are described along with the theory needed to understand them. The tracks used as training and testing scenarios are presented. The result times are shown with the trajectories from the simulation. The results are documented and discussed. Both agents trained on the CIIRC track managed to safely complete a lap with a better time than the Follow the Gap algorithm provided as a baseline. Finally, a few improvements to the simulation model, as well as to both model-free and model-based algorithms, are proposed.

Keywords: End-to-end control, F1TENTH, Autonomous driving, Reinforcement Learning, Race track, Lap time

Supervisor: Ing. Jaroslav Klapálek

Abstrakt

Táto práca sa zameriava na autonómne riadenie modelu vozidla F1TENTH v prostredí pretekov s jedným agentom. Po preskúmaní už existujúcich riešení boli vybrané dve metódy. TD3, algoritmus ktorý nepoužíva model, a Dreamer, algoritmus ktorý sa spolu s reakciami snaží odhadnúť aj model okolia. Opísané sú prostredia pre simuláciu a reálne scenáre experimentov. Metódy sú opísané spolu s teóriou potrebnou na ich pochopenie. Uvádzajú sa trate použité ako tréningové a testovacie scenáre pre experimenty. Sú uvedené časy výsledkov s trajektóriami zo simulácie. Výsledky sú zdokumentované a diskutované. Oba agenti úspešne zvládli prejsť kolo s lepším časom ako Follow the Gap algoritmus ktorý bol použitý pre porovnanie. Na záver sa navrhuje niekoľko vylepšení simulačného modelu, ako aj algoritmov bez modelu a algoritmov založených na modeli.

Kľúčové slová: Riadenie end-to-end, F1TENTH, Autonómne riadenie, Posilované učenie, Pretekárska dráha, Čas na kolo

Preklad názvu: End-to-end riadenie F1/10 autonomného auta s využitím neuronových sietí

Contents

Figures

Tables

Chapter 1

Introduction

As early as the first vehicles were made, people started racing them. People modified their vehicles to be faster in order to achieve faster times on the race tracks. This led to rapid development in that area, far faster than in the commercial mass-produced vehicles. This still holds true, even to this day.

Nowadays, self-driving vehicles are becoming more popular, and the same principle is being applied. More autonomous racing competitions are being created as researchers and engineers are motivated to develop and compare their progress. There are already many kinds of competitions in autonomous racing, from scaled-down models that are more affordable and approachable, like DonkeyCar and F1TENTH [1], to full-sized race cars, like Indy Autonomous Challenge or, recently, the Abu Dhabi Autonomous Racing League. In this thesis, we focus on single-agent racing on a track. We focus on completing a lap while optimizing the lap time.

The current standard for controlling autonomous racing platforms consists of different modules like mapping, localization, and control. The control modules have a direct impact on the lap times achieved. There are multiple control methods with various complexity and adjustability. The Follow the Gap [2] algorithm is a simple reactive algorithm that follows the deepest free space it can see. This algorithm can avoid obstacles, but it cannot navigate through tight turns. Slightly more complex is Pure Pursuit [3], a trajectory follower algorithm. This algorithm is dependent on the given trajectory that needs to be computed. A highly adjustable approach is the Model Predictive Control [4] algorithm. This algorithm solves an optimization problem while utilizing the vehicle model. This leads to its biggest downside: a model needs to be measured and provided. If this model is too simple, the results will not be as good as those of a more complex one. Implementing a complex model is not trivial and, in many instances, impossible as we cannot measure all the information needed to design one. In addition, if the agent gets into an undefined situation, the agent fails, which leads to poor scalability.

Machine Learning approaches like Imitation and Reinforcement Learning provide end-to-end solutions to these problems. If the agent learns from the experience or guidance of a teacher, just like humans, it can resolve unseen situations by utilizing said experience. It also does not need a defined model as it can predict one from interacting with the environment. This approach is still in its early stages compared to other well-established ones. Nevertheless, it is gaining traction as the limits of the classical control methods are approached. Due to the nature of learning from experience, most of the experiments are still held in the simulation environment.

In this thesis, we aim to implement, train, and test two Reinforcement Learning agents. Two approaches are tested, model-free and model-based. The model-free approach chosen is the TD3 [5] algorithm, and the model-based approach is the Dreamer [6] algorithm. The platform chosen for testing is the F1TENTH [1] model race car. To our knowledge, no one tried the TD3 algorithm on a real-life race track at the time of writing this thesis. The Dreamer agent was once successfully tested, and we build on it in this work. We introduce a modification to the algorithms that penalizes slow progress on the track. Both agents are trained in simulation and tested in both simulation and real life. This way, we can show the difference between simulation and real environment, the *sim-to-real gap*.

The second chapter of this thesis reviews and discusses other end-to-end approaches that solve similar problems. The third chapter defines the problem and describes the environment and the platform used. Next, in the fourth chapter, we introduce the theoretical basics to help explain and understand the methods used to solve the given task, after which, in the fifth chapter, we describe the methods used. In the sixth chapter, we provide the experiment scenarios, in the seventh, the results of these experiments are presented and discussed. We suggest improvements for the simulation environment, as well as for both agents, in the eight chapter, future work. The ninth chapter draws the conclusion.

The videos of the real-life experiments, as well as the implementation can be found on: `https://github.com/CTU-F1T/DP-End-to-End-Control`.

Chapter 2

Literature review

This section describes and discusses other approaches to similar tasks. Apart from the possible methods that can be used for end-to-end control, the safety of end-to-end methods, the simulation environments, and real-life platforms are discussed.

2.1 Methods

The main categorization of Reinforcement Learning methods is model-free and model-based. Model-free methods are less complex as only the actions of the agent are trained and predicted. This leads to less computing in one step. Thus, the model-free methods require fewer resources. This, however, also leads to a downside, and that is they extract less information from each interaction with the environment, which is called sample inefficiency. Therefore, they need more experience to learn. Model-based methods are providing a solution. While learning, they also learn the behavior of the environment, which is more sample-efficient. Although that may be impossible in some cases, and in some situations, it is possible, but it requires a lot of computational power.

2.1.1 Model-free

Model-free methods are popular in robotic control, as can be seen in the stabilization of a robot [7], many instances of race car control in [8], or full-sized vehicle control in [9]. While having some success over the years, the biggest known problem is their sample inefficiency.

In vehicle End-to-End control, algorithms such as TD3 [5], DDPG [10], SAC, D4PG, PPO, are often tested [8]. The authors in [11] even managed to show better performance than human drivers, although only in the simulator game Gran Turismo Sport. As almost always in racing, the goal was to achieve the fastest lap possible. The algorithm used here was SAC.

In [7], authors deal with the stabilization and control of a sk80 two-wheeler robot. Their problem is described as a partially observable Markov decision process (POMDP). That can help us define the world from sensors like Lidar as stochastic and not deterministic. This is done to wrap up all the factors that cannot be controlled or learned. Soft Actor-Critic (SAC) algorithm was used. In the preliminary tests, it performed better than TD3 and A2C. It also tackles a few possible improvements like DroQ or D2RL algorithms.

In [12], a basic TD3 algorithm implementation without modifications is used along with a supervisor system. This system keeps the agent safe during online learning, which helps to eliminate the sim-to-real gap. This proves to be working, and it is implemented at slower speeds. However, as a result of these limitations, the performance and speed suffer.

This is tackled by [13]. It is also based on the same implementation of TD3 as [12]. The main idea is to change the reward function from the progress on the track to the difference between the agent's actions and ideal states, given by the optimal trajectory with said states. The article claims a more stable performance with comparable speeds to the base TD3. The minimum lap time was still mostly done by pure TD3 without modifications. However, we argue whether the agent could exploit the physics in the simulation without penalization for doing not plausible actions. With the changed reward, the agent was significantly more successful in testing, as it completed more of the twenty test laps. The average lap time was comparable to the unmodified TD3, although the lap times were more stable and predictable as they had less variation. Another benefit was that this way, the agent could learn faster as the optimal line guide as to where to slow down and where to go full throttle. The disadvantage brought by this is the robustness of the agent. As it learns from the optimal trajectory, the agent is hardly applicable to other tracks. Also, the trajectory must be provided, which requires generating the optimal trajectory, which is a difficult optimization task. This learning and testing was only done and tested in simulation because of concerns about crashing a real-world model car.

Beneficial modifications to the TD3 algorithm are proposed in [14]. They use a priori knowledge to learn an efficient breaking strategy. A heuristic is calculated to modify a reward at the beginning of the learning, and as the learning progresses, the weight of this reward is lowered. Also, an adaptive exploration strategy is utilized to lower the impact of the exploration noise at higher speeds. Lastly, an adaptive action smoothing is added to avoid sudden and strong changes. The reward function uses the road curvature to restrict unnecessary control inputs. These changes have proven beneficial in both safety and lap times.

Often, some modifications are utilized from other machine-learning fields. Multiple adjustments for the DDPG algorithm are tested in [15]. Namely: (i) Window sampling should improve understanding of the current state while using a window of the last n states; (ii) Long Short-Term Memory helps to utilize experience from an arbitrary number of steps; (iii) Multi-step Targets incorporate the next n rewards obtained along the trajectory starting from the current state and following a policy close to the current policy at time step t; (iv) Prioritized Experience Replay focuses on learning efficiency by sampling more frequently more important transitions. This paper shows a general improvement in performance and robustness from the base implementation.

2.1.2 Model-based

Model-based methods try to solve the sample inefficiency by using as much information as possible. Apart from an agent, they also learn the world model from which other predictions can be drawn. They are more complex than model-free methods, and therefore, they require more computational power and more time.

The world model can frequently be clearly separated from the agent, as can be seen in [16]. Their method combines Imitation Learning and Reinforcement Learning, where data is pre-gathered by an expert human driver at the start. The human driver drives with various random noises injected to gather input on behaving in unexpected, often critical, situations. This is done to provide safety and better sample efficiency. In training, human actions are weighted less as the agent learns to drive and minimize the loss function. The network used is RestNet18 [16]. It is divided into policy network and predictive world model. The world model uses multiple gated recurrent units to predict and decode state, speed, and collisions. The problem is defined as POMDP, and the main data input is an RGB picture from the camera.

At Wien Technical University, a model-based approach with Dreamer network architecture [6] was successfully tested [17]. As with other model-based approaches, it focuses on sample efficiency. Learning the world model utilizes so-called latent imagination to create and simulate training sequences without interacting with the environment. This method was tested on a real model race car [17].

An attempt to implement DreamerV2 [18] to a race car was made by [19]. They modified the network by not only predicting the actions in states but rather planning the whole trajectory with an expectancy value. The experiments in simulation have proven a marginal improvement over prior work [17] with a more efficient learning process and more robust agent overall. However, this was not tested on a real car.

2.2 Safety

Safety is a big concern in RL in vehicle control, as the trained agents behave like a black box, and we cannot track down the steps of how they made a decision. It is one of the main reasons that most agents are only tested in simulation and games, as can be seen in the examples provided by [8]. Safety and reliability are tested in multiple environments in [20]. First, algorithms TD3 and DDPG are trained and tested in a simulation, where both algorithms drive with similar run times and success rates. The next experiment is a real-world track, modified not to have reflective surfaces. They show that the TD3 agent is more robust in the real world. It completes more runs and behaves more consistently than DDPG. The last environment is on the same track but without surface modifications. In an unmodified environment with reflective surfaces that are problematic for Lidar rays, TD3 performed almost the same, while DDPG crashed in most runs. Overall, they show that

DDPG is less robust and it handles the sim-to-real gap worse than the TD3 algorithm.

2.3 Online Learning

Another way to ensure safety and overcome the sim-to-real gap is to train the agents directly on the car in the real-world environment. With online learning, supervision is needed to protect the car from destroying itself in the process. The most straightforward method is to have a human supervisor overseeing the learning [9]. This is done on a regular-sized vehicle on real roads. A simple reward function for distance traveled with the DDPG algorithm is used.

Another way of supervision is used in [12]. A safety kernel is generated iteratively. If the car goes into the restricted, *unsafe zone*, the control is handed over to a classic control algorithm, and a penalty goes to the agent as if it crashed. The agent is then driven to a safe state, and the learning is resumed. In the experiments, the car drives for 2000 steps, which is about 20 seconds, to gather data, then stops to calculate the network updates, as it is impossible to do the calculation while driving the car autonomously.

A framework labeled FASTRLap is introduced in [21]. Several optimizations in learning are introduced to ensure that agents can learn to drive reliably in about 10 to 20 minutes. One of the ways to achieve this is by introducing a phase before online reinforcement learning. This phase uses Offline Reinforcement Learning with Implicit Q-Learning (IQL) to extract a critic for a readily available, diverse offline dataset collected on a different robot, using a similar task objective: goal-directed velocity toward checkpoints selected from a mix of future states and random points in space [21]. After that, the critic is discarded, as only the image encoder is extracted. This encoder is optimized for extracting task-relevant data from an image. Multiple experiments in different environments, including off-road ones, are presented.

2.4 Environments and platforms

As hardware in robotics and vehicle control is quite expensive, most of the time, the learning and testing are done in simulation, just as in most cases in this survey [8]. There are some efforts made to control real cars, as highlighted in [9, 12, 17, 20]. In a virtual environment, many simulators with gymAPI are popular [8], like F1TENTH Gym [12, 13], the Roborace Simulator, the SVL Simulator, TORCS [14] or Racecar Gym [17]. Besides simulators, simulation videogames like Gran Turismo Sport can be used [11]. The simulator chosen for this work is the Racecar Gym [22] because it uses a more complex physical engine rather than just using a simple kinetic model as in F1TENTH Gym [23]. Other more complex simulators or games like TORCS, Carla, or Gran Turismo Sport are, on the other hand, either computationally too demanding or too difficult to work with software-wise.

There are also multiple real-life model racing series and platforms that are useful in research. An F1TENTH platform [1] is a popular autonomous vehicle control research choice. It is a versatile platform that can be used both on a track and in an off-road environment [21]. It can be outfitted with lidar [12, 17, 21] or camera [16, 21] for vision and data gathering. Other popular platforms are AutoRally or DonkeyCar [8]. In [9], they even controlled a full-sized vehicle on normal roads.

Chapter 3

Problem statement

In this chapter, the problem, as well as the simulation environment and the platform for real experiments, are defined. All of the parameters used are described.

3.1 Problem statement

The goal of this thesis is to train and compare model-based and model-free agents to complete a lap on a race track successfully. The agents are trained on multiple tracks in a simulation environment and then tested in simulation and on an F1TENTH platform in a real environment. In a real-life scenario, the classic control agents will be used for comparison.

The problem of driving on the track can be defined as a Partially Observable Markov Decision Process (POMDP). It is a tuple of (S, A, Ω, O, T, R), where

- S is a set of states,
- A is a set of actions,
- Ω is a set of observations,
- O is a stochastic observation function, $(O : S \times \Omega \to [0, 1])$ returning the probability of perceiving an observation,
- T is a stochastic transition function, $(T : S \times A \times S \to [0, 1])$ returning the transition probability between two states by applying actions in a given state,
- R is a reward function, $(R : S \times A \times S \to \mathbb{R})$ returning the numerical reward assigned to a transition.

In our case, the observation is a lidar scan with 1080 distance point measurements in the range $[0.06\,\mathrm{m}, 10\,\mathrm{m}]$. As the only sensor and observation utilized, it forms the observation space. The used lidar is only a single plane, measuring distances only within a certain plane that is parallel to the floor. Thus, the track's boundaries should be at least as tall as the distance of lidar from the ground to be detected. A set of actions, or action space, is a two-dimensional vector where the first element controls the electric motor, and the second is the steering angle. Both elements are in the range [-1, 1].

3.2 Training environment

The simulator chosen is Racecar Gym [22]. It is made for a miniature racecar, like F1TENTH, using the bullet physics engine with Pybullet [24]. It provides a Gymnasium [25] interface, which is an API standard for single-agent reinforcement learning environments. The sensors provided by the simulator are: (i) pose, (ii) velocity, (iii) acceleration, (iv) lidar, (v) RGB camera. The interface to access the observations from the sensors:

Key	Description
pose	Holds the position (x, y, z) and the orientation (roll, pitch, yaw) in that order.
velocity	Holds the x, y, and z components of the translational and rotational velocity.
acceleration	Holds the x, y, and z components of the translational and rotational acceleration.
lidar	Lidar range scans.
rgb_camera	RGB image of the front camera.

Table 3.1: Observation interface [22]

As stated, only the observation from the lidar scan is utilized in order to minimize dependency on other measurements that are either not guaranteed or may be inaccurate on the real race car. It provides 1080 rays, distributed evenly over 270°. The parameters are set to match the lidar on our car, so the maximum range is 10 meters, and the minimum is set to 10 centimeters. Control inputs, or actions, control either the motor or speed and the steering angle of the wheels.

Key	Description
motor	Throttle command. If negative, the car accelerates backward.
speed	Normalized target speed.
steering	Normalized steering angle.

Table 3.2: Controls interface [22]

Since using both ways to control speed is unnecessary, control of the motor is used as it is more intuitive and similar to normal vehicle control. Both values are one-dimensional and normalized in an interval $[-1, 1]$. The motor controls are mapped to -1 as a full brake and 1 as a full throttle. The steering is mapped to -1 as full left and 1 as full right. While additional information about the environment and agent state is available, it is not used as it could lead to reliance on data that the agent will not have in a real-world environment.

3.3 F1TENTH platform

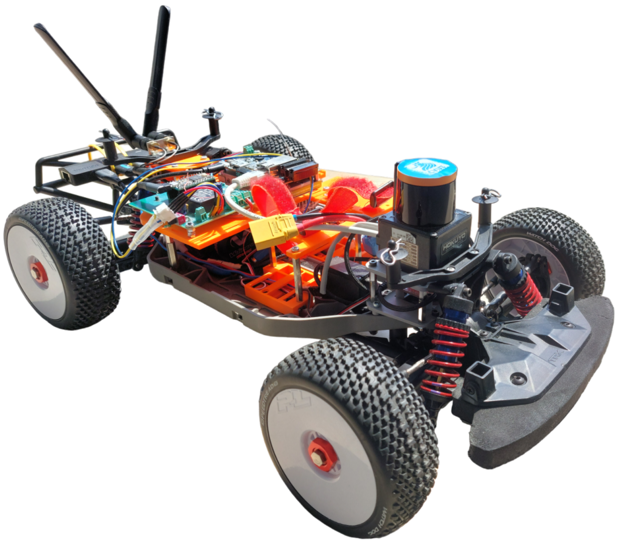


Figure 3.1: The F1TENTH build used in this work (without the batteries).

A 1:10 scale race car, *F1TENTH* [1], is used in real-life experiments and as a model for the simulator. F1TENTH race cars are 1/10th the size of real F1 vehicles and are used as a test-bed for autonomous algorithms [8].

Part	**Type**
Electric motor	Velineon 3500 BLDC
Servo motor	Traxxas 2075R
Lidar	Hokuyo UST-10LX
Inertial Measurement Unit	SparkFun 9DoF Razor IMU
Onboard Computing Unit	NVidia Jetson TX2

Table 3.3: F1TENTH components used.

This platform is all-wheel drive with front-wheel steering. It is driven by the electric motor using a Vedder Electronic Speed Controller (VESC). The servo motor is used for steering. As for the sensors, our build provides lidar, which is utilized in this work. It runs on 40 Hz and provides a 270° range with 0.25° resolution (1080 segments). Besides the lidar, an inertial measurement unit is provided but not utilized. The software providing an interface to everything is ROS2, which runs on the onboard computing unit.

Description	**Symbol**	**Value**	**Unit**
Vehicle width	W	0.295	m
Vehicle length	L	0.535	m
Track-width	W_t	0.253	m
Wheelbase	L_{wb}	0.330	m
Center of gravity to the front wheel axis	L_f	0.165	m
Center of gravity to the rear wheel axis	L_r	0.165	m
LiDAR angular detection range	θ_{ADR}	270	°
Maximum steering angle	δ	0.35	rad

Table 3.4: Measurements of the F1TENTH platform

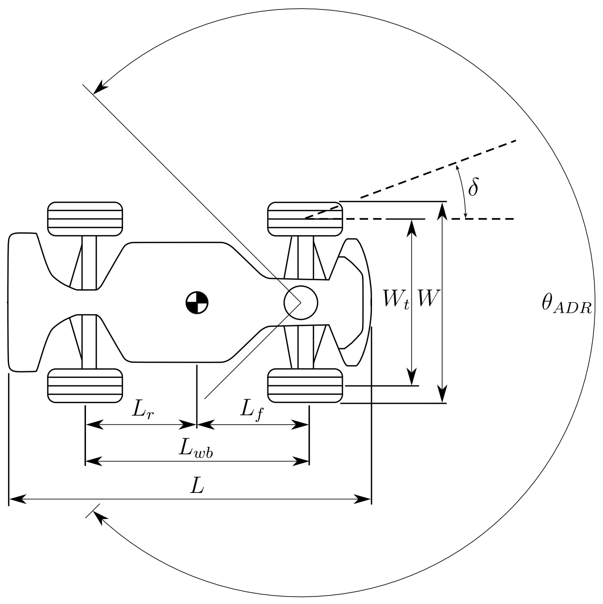


Figure 3.2: F1TENTH platform top view.

Chapter 4

Neural networks in control theory

This chapter defines and explains the terms and the theory used in this thesis. Section 4.5 is also a brief introduction to Reinforcement Learning and its principles, as well as the methods that build on those principles.

4.1 Modular and classic control approaches

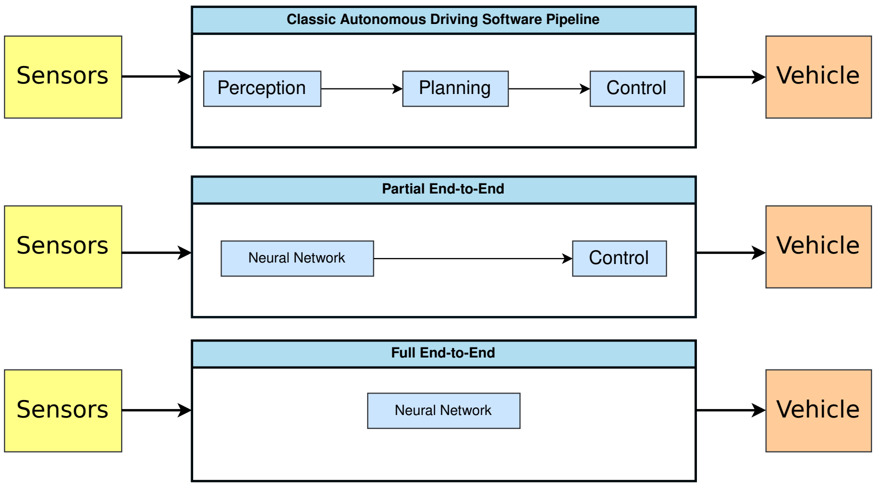


Figure 4.1: Classic autonomous driving software pipeline in comparison to partial and full end-to-end software pipeline, based on [8].

Modular approaches consist of software modules with focused tasks combined into a pipeline. This separation into modules led to the development of software frameworks such as Robot Operating Systems (ROS). This way, the development of complex control systems can be streamlined, as the research can be focused on one isolated task. The independent modules can be used in compatible systems, not only the one they were developed on.

However, modular systems have a number of disadvantages. Different situations regularly require different modules to be implemented. This leads to poor scalability as implementing and developing new modules to approach new situations is time-consuming and complex. Furthermore, not all of the information is utilized. The data used is explicitly defined. Thus, the implementation overlooks some implicit connections in the system that would help to control it.

4.1.1 Pure pursuit

Pure pursuit [3] is a trajectory-following algorithm. It is widely known and has been in use for years now. The algorithm calculates the curvature that will move a vehicle from its current position to some goal position. The algorithm chooses a goal position some distance ahead of the vehicle on the path. The name is an analogy to the algorithm, as it literally pursues the point chosen on the trajectory.

4.1.2 Model predictive control

Model Predictive Control [4] (MPC) is a more complex approach used to follow a given trajectory. The approach is an optimization method to select control inputs by minimizing the objective function. MPC does this while satisfying given system constraints and working in a finite time horizon. This approach can be configured by changing the objective function and adding system constraints to achieve the desired results. However, the general downside is greater computation cost and difficult fine-tuning.

4.1.3 Follow the gap

The Follow the Gap (FTG) algorithm is a reactive algorithm that maximizes the gap between obstacles while moving toward the goal. The algorithm constructs a gap array around the vehicle and calculates the best heading angle for heading the agent into the center of the maximum gap ahead [2]. The algorithm simultaneously considers the goal point. The following of the largest gaps is connected to goal tracking by a fusion function.

4.2 Partial End-To-End

As seen in Fig. 4.1, any part of the pipeline can be swapped with a Machine Learning (ML) model trained for that particular task. For example, this can be beneficial when defining a model we cannot represent analytically. A model that predicts the behavior close enough to the real-world counterpart may be better than one that simplifies things by relaxing the constraints and adding unreal predicates. Neural networks can also help by encoding high-dimensional inputs into low-dimensional representations. This can be particularly helpful when dealing with an image input from a camera. Classic controllers can then use this representation further as input.

4.3 End-To-End

In end-to-end driving, the entire pipeline is swapped for a neural network. End-to-end approaches transform sensor inputs to driving commands, and they are treated as a single learning task [26].

The main theoretical advantage is that it can learn optimal control with the right network configuration and with enough expert driving data for imitation learning or enough environment interactions for reinforced learning.

With no explicitly predefined limits on the information, end-to-end models can utilize all the data they can gather from the environment. The ML approach allows an indirect and implicit way of reasoning for the model, which is not possible with the modular way of control. This approach has proven useful in many different fields, like object recognition and detection or natural language processing. Closer to the problem in this thesis are examples that show superhuman-level performance in ATARI video games [6, 18, 27] and high-level performance in more complex competitive games like Starcraft II [28], or DOTA 2 [29]. The complexity of such dynamic tasks in continuous spaces shows that there is a reason to research vehicle control using such approaches.

End-to-end control, however, has some flaws in practical uses. First of all, the assumption of the ability to optimally control, predict, or solve a problem in general in an infinite horizon is not manageable. It is not possible as we cannot provide an infinite amount of data or experience. The learning process itself is strongly dependent on initial conditions and configuration of hyperparameters. A slight change in both of them may result in a significant change in the results. That leads to an issue with interpretability and predictability.

The transformation from the sensors' outputs to the driving inputs is not transparent and fully understandable. Simplistically, we can see it as a very complicated, deterministic, non-linear mathematical function with many parameters and weights. With no intermediate outputs, it is much harder to trace the cause of an error. It is also very hard to explain the observed behavior and the decisions taken [26].

4.4 Imitation Learning

Imitation Learning (IL) is a subset of Supervised Learning. In Supervised Learning, the agent is presented with labeled training data. The agent learns to process the data to produce the desired label or behavior. In IL, the training dataset is composed of task executions by a demonstration teacher [30]. In autonomous vehicle control, the expert is a human driver, and the mimicked behavior is control of the vehicle [26]. This includes steering, acceleration, and braking.

Alternatively, a control algorithm can be used as the teacher. The agent tries to mimic and optimize its actions to imitate the expert driver or control algorithm. This way of learning works well in simple tasks and scenarios, like following a lane. However, in more complex situations where we do not have the training data, this method generally fails. This leads to a Distribution shift problem [31].

4.4.1 Distribution shift problem (and its solutions)

Since the actions are not copied exactly but rather learned from an example, the driving observations differ from the states given by the expert driver. This should lead to different actions where the agent lacks the required training data. Hence, it is not prepared for unseen situations. For example, if the agent learns from a driver who prefers staying in the middle of the road, it might not react correctly and safely if the car goes to the edge of the road or track. The model might not recover from an unseen state and return to the middle of the road. Some ways of dealing with such problems are: (i) modifing already existing data (data augmentation), and (ii) modifing the collecting process itself (data diversification).

Data augmentation. Data augmentation [32] is useful mostly while using a camera input. Images can be, e.g., cropped, blurred, or rotated. The artificial images must be associated with target driving commands to recover from such deviations. This method has been sufficient to avoid accumulating errors in lane-keeping [26].

Data diversification. Data diversification aims to collect more diverse data by adding different noises to the data-gathering process. This shows the model how to behave in strange situations by putting the driver in those situations. In [16], the data is pre-gathered with an expert human driver to help with the first part, imitation learning. The human driver drives with various random noises injected to gather input on behaving in unexpected, often critical, situations. This is done to provide safety and better sample efficiency.

On-policy learning. Another way of tackling the distribution shift problem is using on-policy learning. This method gathers data while the model is driving. That means the expert gives online commands to the agent during the execution of the policy. This concept was proposed by DAgger [33]. This leads to control situations that would not occur under normal human driving. Nonetheless, it requires a human observer during the learning process, and the learning must be slowed down so that the human can observe it.

Learning from an algorithm. The alternative to a human expert as a driver is to use a control algorithm as the teacher. However, this leads to other problems. Using a standard algorithm as a teacher means the agent would not outperform it. If the agent gets into a state that is unsolvable for the algorithm, it will get a bad, unreasonable command from the algorithm that it will try to mimic, which can negatively affect the model as a whole. Another problem is the use of a machine learning agent for states that a classic control has already solved.

This thesis does not use Imitation Learning as it needs an expert driver, and gathering the required data is time-consuming. Also, it has been stated that reinforcement learning converges to better results as the immediate reward and environment and action space [26].

4.5 Reinforcement Learning

Definition with a brief summary from [34] states that Reinforcement Learning (RL) is trying to map situations to actions to maximize a quantifiable reward signal. The learner is not given which actions to take but must discover which ones yield the most reward by trying them. In the more complex and challenging cases, actions may affect not only the immediate reward but also subsequent future rewards. The two most important distinguishing features of RL are training without labeled actions to learn from and delayed rewards that would carry over the consequences of actions. The inputs and outputs do not differ from the ones in imitation learning, apart from the driving data that are not needed since everything is learned from experience. Both approaches, IL and RL, can be combined. IL is used for fast learning the basic controls at the start, with RL being used for fine-tuning and driving optimization as in [16, 21].

4.5.1 Reward signals

A significant difference from imitation learning is that the agent has no guidance to follow as it only relies on reward signals from the interactions with the environment. Consequently, the choice of positive and negative rewards greatly influences the learning process. Making these too complicated is not beneficial as it might lead to unreadable and unpredictable behavior. Simple reward signals help clearly define the purpose and limit unpredictable behavior. However, it can also be beneficial to consider more complex rewards to define what is desirable explicitly if the task at hand requires it. A commonly used reward signal in autonomous driving on a race track is simply the progress through the track with a bigger bonus when a lap is completed [12]. A slightly more complex reward can be seen in [13], where the rewards are based on the distance to the optimal race-line states to minimize lap times. The positive reward is then:

$$r = 1 - |v_{agent} - v_{optimal}| - |\delta_{agent} - \delta_{optimal}|, \tag{4.1}$$

where v $[\mathrm{m} \cdot \mathrm{s}^{-1}]$ is velocity and δ [rad] is steering angle. The behavior is further reinforced by adding a $+1$ reward for completing the lap.

Combining multiple reward signals can be beneficial in supporting a desired behavior. The positive rewards for speed, progress, and lap times are enough in theory but, in practice, are often reinforced by negative rewards for collisions [13, 26] or getting into restricted zones [12]. This promotes safety and consistency.

Collisions. The most common negative reward, or penalty, for a collision is a large negative constant. A more complex way of discouraging collisions and promoting safety is used in [11, 35]. Their solution is to multiply the negative constant for crashing by the kinetic energy generated by the crash:

$$E_K = \frac{1}{2} \cdot m \cdot v^2, \tag{4.2}$$

which promotes the idea that minimizing the damage by slowing down as much as possible is better than colliding at full speed if a collision is unavoidable. As the weight m [kg] does not differ much in the same category for racing, the only controllable part is the velocity of the vehicle v. This is reflected in the reward signal:

$$r_t = r_t^{prog} - \rho_w \cdot c_w \cdot \|v_t\|^2, \tag{4.3}$$

where t stands for time step, ρ_w is a binary variable that indicates a crash (1 = crash), and c_w is a constant penalty for hitting a wall.

Overtaking. Overtaking is another task that a reward signal can define. It is a substantially more complex problem than just driving alone on the track as fast as possible. Authors in [35] propose a solution to overtaking. Firstly, they define a basic reward as a difference in progress in the state s_t at the time t rather than progress at the moment:

$$r_t^{racing} = (cp(s_t) - cp(s_{t-1})) - \rho_w \cdot c_w \cdot \|v_t\|^2, \tag{4.4}$$

where $cp(s_t)$ stands for projection on the central line in time step t, $t-1$ indicates the previous time step and the difference between the two projections is the distance traveled in t. That encourages the agent to progress in each time step. The reward that aims to overtake a single opponent i is defined as:

$$\begin{aligned} r_t^i &= \rho^i \cdot c_r[\Delta cp(s_{t-1}^i s_{t-1}^k) - \Delta cp(s_t^i, s_t^k)] \\ \Delta cp(s_t^i, s_t^k) &= cp(s_t^i) - cp(s_t^k) \\ \rho^i = \rho(s_t^i, s_t^k) &= \begin{cases} 1, & |\Delta cp(s_t^i, s_t^k)| < c_d \\ 0, & \text{otherwise,} \end{cases} \end{aligned} \tag{4.5}$$

where k represents the agent's car controlled by the learning policy, cd is a hyperparameter for the detection range, and hyperparameter c_r is a trade-off between the aggressiveness of the overtaking maneuver and the safety of the drive. The whole reward signal is defined as:

$$r_t^{overtaking} = r_t^{racing} - c_c \cdot \rho_c \cdot \|v\|^2 + \sum_{\forall i \in C \setminus \{k\}} r_t^i, \tag{4.6}$$

where C is a set of all cars on the track, c_c is a constant for crashing with another car, and ρ_c is a binary flag for collision with another car.

4.5.2 Model-free and model-based algorithms

We distinguish two types of RL based on what the RL agent is trying to learn: (i) model-free, where only the agent and its actions are learned, and (ii) model-based, where the environment model is also learned. As described by [36]: "A model-based algorithm is simply any algorithm that makes use of the transition dynamics of an environment, whether learned or known in advance. Model-free algorithms are those that don't explicitly make use of the environment transition dynamics."

Model-based algorithms. In general, model-based algorithms have marginally better sample efficiency [17, 34, 36]. This is due to extracting information not only about the agent itself but also about the environment and its transitions. A perfect model is beneficial as it can simulate interactions with the environment without actually acting in it, as in [6]. The model can give foresight to the agent as it predicts how the environment will behave, which is very helpful in finding the optimal action or strategy.

While all of this sounds promising, model-based methods have some drawbacks. For most problems, learning or modeling the transitions in the environment is not straightforward. Many systems have stochastic transitions, whose dynamics are not known precisely [36]. In that case, a model needs to be learned as the probabilities cannot be modeled in a normal way. The model-based methods are still in relatively early stages of development and still face many challenges [36].

Model-free algorithms. Model-free methods are simpler than model-based ones. While this compromises sample efficiency, the computation requires fewer resources. The fact that no model is required means that these methods can be used in almost any environment and applied to any task. The agent is learning only its own values and predicting strategy and actions.

4.5.3 Temporal difference learning

Temporal difference (TD) methods combine ideas behind Monte Carlo methods and dynamic programming (DP). "Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome." [34]

In general, TD methods use experience, or interaction with the environment, to solve policy evaluation. Non-terminal states are evaluated from experience in every time step t with the current estimate of current state $V(S_t)$, the observed reward R_{t+1}, and the current estimate of the next state $V(S_{t+1})$:

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot [R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t)], \tag{4.7}$$

where α is a constant step size parameter, and γ is a parameter for discounting future rewards. This is known as TD(0), a base for temporal difference learning algorithms. The whole algorithm is described in Algorithm 1.

Algorithm 1: Tabular TD(0) for estimating v_π [34]

```
Input: the policy $\pi$ to be evaluated
Initialize $V(s)$ arbitrarily (e.g. $V(s) = 0, \forall s \in \mathcal{S}^+$)
for each episode do
    Initialize $S$
    while $S$ is not terminal do
        $A \leftarrow$ action given by $\pi$ for $S$
        Take action $A$, observe reward $R$, and next state $S'$
        $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$
        $S \leftarrow S'$
    end
end
```

4.5.4 On-policy and Off-policy Learning

The model-free algorithms are further divided into on-policy and off-policy methods. A policy is a set of rules for an agent to determine its actions at a given state, and it is also divided into *behavioral policy* and *target policy*. The behavioral policy determines which actions are taken to navigate the environment and is used for exploration. The target policy optimizes the decision process, maximizing the reward or objective function.

The main difference between On-policy and Off-policy learning is that in an Off-policy algorithm, these two can differ. The On-policy algorithms optimize a chosen policy q^π by evaluating only the move chosen by the policy. The Off-policy methods find the optimal policy q^* because their target policy is always greedy, which means the highest possible reward is always selected, and other possibilities for actions in current states are ignored. An explanation is shown in the cliff walk example [34]. On-policy leads to a safer and longer route, while Off-policy prioritizes finding the optimal value without regard for safety.

4.5.5 Q-Learning

Q-Learning is a model-free off-policy Temporal-Difference control algorithm introduced in 1989 [37]. A single step is defined by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot [R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]. \tag{4.8}$$

The action-value function Q is learned iteratively and updated at every step of each episode. The Q function directly approximates q^*, the optimal action-value function. To ensure the optimal solution and convergence are reached, visiting and updating all state-action pairs is necessary. The full algorithm is in Algorithm 2.

Algorithm 2: Q-Learning: An off-policy TD control algorithm [34]

```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(s_{terminal}, \cdot) = 0$
for each episode do
    Initialize $S$
    while $S$ is not terminal do
        Choose $A$ from $S$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$
        $S \leftarrow S'$;
    end
end
```

The stochastic ϵ-greedy method of picking action is often used to ensure that all states will be visited. This method chooses the best action with the highest value with probability $(1-\epsilon)$. Otherwise, it chooses an action at random. This algorithm has proven optimality in the infinite horizon [37].

4.5.6 Actor-critic methods

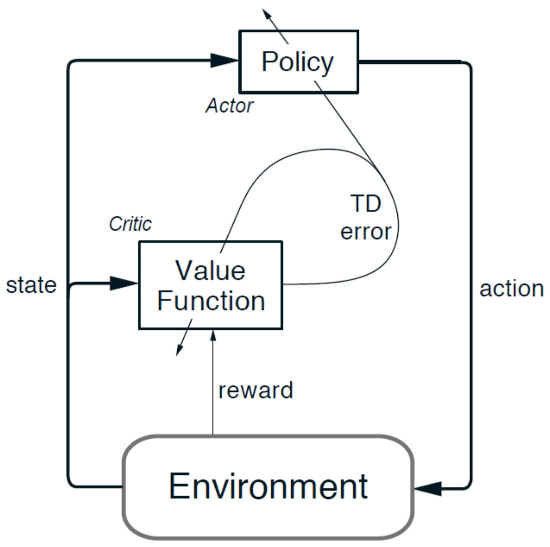


Figure 4.2: The actor-critic architecture, taken from [34].

Actor-critic approaches are based on temporal difference methods with distinct memory architectures separating the policy from the value function [34].

Labeled as the actor, the policy structure operates autonomously in action selection, while the critic, denoting the estimated value function, rates the actions the actor executes. This learning method is always on-policy as the critic learns and evaluates the current policy followed by the actor. The evaluation drives all the learning in both the actor and the critic. This value is the sole output from the critic in the form of a single scalar signal.

4.5.7 Deep Reinforcement Learning

Deep Learning (DL) is widely used as it can solve difficult and complex tasks in many different areas. It has proven it excels at complex nonlinear function approximation over the years. DL is commonly used for image and object recognition, processing and generating speech, or largely popular Natural Language Process (NLP) models like GPT or LAMA. The principles behind DL can be used in RL, as the function approximation can be used as a value or policy function. Also, the representation used in Deep Neural Networks helps with scalability and high-dimensional spaces [38]. This leads to better scalability. The result of combining DL and RL is labeled Deep Reinforcement Learning.

Latent space. One way to represent spaces internally in DRL is latent space. Regularly, the observations and spaces are large and high dimensional. This can lead to a lot of parameters being required to represent even smaller and simpler things. In machine learning, latent spaces are a popular way to solve this issue. In [39], latent space is defined as: “a high-dimensional vector space in which each data item is represented as a single vector”. It encodes meaningful data in an internal, more compact representation.

4.5.8 Transfer from simulation to real world

The *sim-to-real gap* is a big obstacle in autonomous driving. It is the difference between the simulation environment and the real world. This difference shows itself in the agent’s behavior, which receives a different response in the simulation environment than in the real one. This leads to different transitions from the ones the agent learned in the simulation.

This is even more apparent in DRL, where the whole end-to-end algorithm is sort of a black box. In classical control, this problem can be mitigated by setting the parameters for the algorithm used. However, in reinforcement learning, you can’t change the learned functions. You may try to re-train them in the real world [9, 12], but that brings many complications in itself.

The agent training on a real vehicle or model needs supervision to prevent the vehicle’s destruction, e.g., a human driver to intervene when the agent gets to a dangerous state is used in [9]. A classic control algorithm can also be used with some form of definition and detection of a dangerous state. This is utilized in [12]. Here, the track is divided into safe and unsafe regions. This *safety kernel* lists recursively safe states. This ensures that every safe state leads to at least one safe state. However, this method sacrifices performance and lap time as the exploration space is heavily reduced, and the agent cannot explore enough spaces to get a reasonable driving pace.

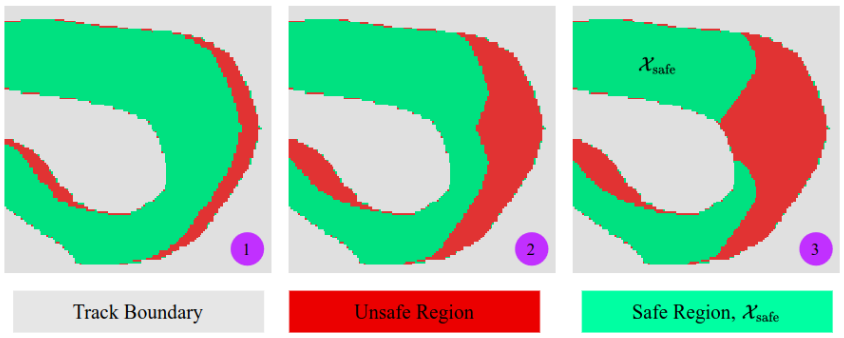

$\mathcal{X}_{\text{safe}}$
1
2
3
Track Boundary
Unsafe Region
Safe Region, $\mathcal{X}_{\text{safe}}$

Figure 4.3: Kernel generation [12]

In Fig. 4.3, we can see that in a particularly dangerous curve, the DRL agent might not have any space to learn. If the agent goes into the unsafe zone, a classical control algorithm will take over to ensure safety, and the learning will get a signal as if the agent crashed.

Chapter 5
Methods

In this chapter, we describe the algorithms that we use to control the car. The chosen model-free algorithm is TD3 [5, 40]. As it is based on and tightly connected to the DDPG [10, 41] algorithm, the description of this algorithm is also provided. For the model-based agent, the Dreamer [6] was chosen as it was already tested in [17].

5.1 Model-free methods

As mentioned in Section 4.5.2, the model-free methods are focused on learning the agent's actions based on the current observed state. It does not learn the transitions in the environment.

5.1.1 Deep Deterministic Policy Gradient

In [10], Deep Deterministic Policy Gradient (DDPG), a model-free off-policy algorithm, is described as "an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy." It builds on the foundation of all off-policy methods. If an optimal action-value function $Q_*(s_t, a)$ is known, then an optimal action $a^*(s_t)$, in the state s_t, can be found at any given time t by solving the following optimization problem [34]:

$$a^*(s) = \arg\max_a Q^*(s, a) \tag{5.1}$$

DDPG combines learning an approximator to $Q^*(s, a)$ with learning an approximator to $a^*(s)$. The algorithm is adapted to be used in continuous action spaces, so it is suitable for use in control and robotics [7, 8]. This adaptation can be mainly seen in the computation of the maximum for actions in $\max_a Q^*(s, a)$.

In the discrete and finite action spaces, we can use a *tabular approach* and compute the Q-values for each action separately and then compare them. In continuous space, the Q-values cannot be computed directly. Solving the optimization problem can be difficult and computationally too expensive. it would need to be solved every time an action is taken by the agent [10].

Algorithm 3: Deep Deterministic Policy Gradient [41]

```
Randomly initialize critic network $Q_\phi(s, a)$ and actor $\mu_\theta(s)$ with
  weights $\phi$ and $\mu$
Initialize target network $Q'$ and $\mu'$ with weights $\phi \leftarrow \phi'$, $\theta \leftarrow \theta'$
Initialize replay buffer $R$
for episode = 1, M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    for t = 1, T do
        Select action $a_t = \mu_\theta + \mathcal{N}_t$ according to the current policy and
          exploration noise
        Execute action $a_t$ and observe reward $r_t$ and new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_t + 1)$
        Randomly sample a batch of $N$ transitions, $\{(s, a, r, s_{i+1})\}$
          from $R$
        Compute targets:
        $y_i = r + \gamma Q_{\phi'}(s', \mu_{\theta'}(s'))$
        Update $Q$-function (critic) by minimizing the loss:
        $L = \frac{1}{N}\sum_i (y_i - Q_\phi(s, a))^2$
        Update the policy (actor) by one step of gradient ascent:
        $\nabla_\phi \frac{1}{N}\sum_i Q_\phi(s, \mu_\theta(s))$
        Update the target networks:
        $\phi' \leftarrow \rho\phi' + (1 - \rho)\phi$
        $\theta' \leftarrow \rho\theta' + (1 - \rho)\theta$
    end
end
```

The Q-learning

One of the conditions is that in continuous action space, the function $Q^*(s, a)$ is differentiable with respect to the action argument [10]. This allows the usage of efficient, gradient-based policy learning $\mu(s)$, which is also used to replace costly optimization of $\max_a Q(s, a)$ with an approximation $\max_a Q(s, a) \approx Q(s, \mu(s))$.

The approximation of $Q^*(s, a)$ is based on the Bellman equation:

$$Q^*(s, a) = \mathop{\mathbb{E}}_{s' \sim P}[r(s, a) + \gamma \max_{a'} Q^*(s', a')]. \tag{5.2}$$

Let's assume that the approximation is a neural network $Q_\phi(s, a)$, with parameters ϕ, and let's also assume we acquired a set of transitions $\mathcal{D}$, that consists of (s, a, r, s') [10]. This tuple describes time step i that consists of state s, the taken action a, the obtained reward r, and the resulting next state s'. This allows the setup of a mean-squared Bellman error (MSBE) function, which is used to approximate the error of Q_ϕ to the optimal Q^* from the Bellman equation:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s')\sim\mathcal{D}}{\mathbb{E}} \left[\left(Q_{phi}(s,a) - (r + \gamma \cdot \max_{a_{i+}} Q_\phi(s', a')) \right)^2 \right] \tag{5.3}$$

DDPG and variants of DQN are Q-learning algorithms for function approximators that are largely based on minimizing the MSBE loss function. There are two main ways DDPG can achieve this. The first one is *replay buffers.*

Replay buffers. They are used to store the already mentioned set of previous experiences, $\mathcal{D}$. Balancing the length of this is a part of the fine-tuning of hyperparameters. A buffer that is too short leads to overfitting, while too long slows down the learning too much. Since DDPG is an off-policy algorithm, all experiences from the buffer are used, even from an outdated policy.

Another big part of minimizing the MSBE is the *target networks.* In DDPG, the target for the Q-function is:

$$Q_{\phi targ} r + \gamma \cdot \max_{a'} Q_\phi(s', a') \tag{5.4}$$

The target depends on the parameter ϕ, which is also being trained. That leads to unstable minimization of MSBE [10]. This is solved by introducing another set of parameters, ϕ_{targ}, which approaches ϕ but with a delay. It is the second target network delaying the first. This network is updated once per main network update by *polyak* averaging:

$$\phi_{targ} \leftarrow \rho\phi_{targ} + (1-\rho)\phi \text{ [10]} \tag{5.5}$$

where ρ is a hyperparameter between zero and one that weighs the averaging. Usually, it is set closer to one.

The maximum over the actions is computed from the target policy network, which maximizes $Q_{\phi targ}$. The target policy is then found by polyak averaging the policy parameters over the course of training [10].

To conclude, the Q-learning part in DDPG is performed by minimizing the MSBE loss with stochastic gradient descent:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s')\sim\mathcal{D}}{\mathbb{E}} \left[\left(Q_\phi(s,a) - (r + \gamma \cdot Q_{\phi_{targ}}(s', \mu_{\theta_{targ}}(s'))) \right)^2 \right] \tag{5.6}$$

where $\mu_{\theta_{targ}}$ is the target policy.

The Policy

The objective of the algorithm is to learn a deterministic policy $\mu_\theta(s)$ which gives the best action to maximize $Q_\phi(s,a)$. Again, as we consider a continuous action space, differentiability with respect to action is presumed. With this presumption and with respect to the policy parameters only, a gradient ascent is applicable to solve:

$$\max_\theta \underset{s\sim\mathcal{D}}{\mathbb{E}} [Q_\phi(s, \mu_\theta(s))] \tag{5.7}$$

The Q-function parameters are treated as constants in the policy learning part.

Learning

The policy is trained in an off-policy way. Because in case that the policy is deterministic and the agent tries to explore the environment on-policy, it will not visit enough states with wide enough variety to get useful reward signals. To support exploration, noise is added to the actions in learning. In the later stages of the training, the noise might be reduced to support higher-quality data.

5.1.2 Twin Delayed DDPG

The Twin Delayed DDPG is an algorithm based on the DDPG. Also similarly, it is a model-free off-policy algorithm. Twin Delayed DDPG (TD3) addresses instability and low robustness with respect to hyperparameters and fine-tuning. In [40], authors the issues, “A common failure mode for DDPG is that the learned Q-function begins to dramatically overestimate Q-values, which then leads to the policy breaking because it exploits the errors in the Q-function.”

Algorithm 4: Twin delayed DDPG [5]

```
Initialize critic networks $Q_{\theta 1}$, $Q_{\theta 2}$, and actor network $\pi_\phi$ with random
  parameters $\theta_1$, $\theta_2$, $\phi$
Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1$ to $T$ do
    Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma)$
      and observe reward $r$ and new state $s'$
    Store transition tuple $(s, a, r, s')$ in $\mathcal{B}$
    Sample mini-batch of $N$ transitions $(s, a, r, s')$ from $\mathcal{B}$
    $a^{\sim} \sim \pi_\phi(s') + \epsilon$, $\epsilon \sim clip(\mathcal{N}(0, \sigma^{\sim}), -c, c)$
    $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}$
    Update critics $\theta_i \leftarrow \arg\min_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
    if $t$ mod $d$ then
        Update $\phi$ by the deterministic policy gradient:
        $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
        Update target networks:
        $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
        $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
    end
end
```

One of the improvements over the DDPG is *Clipped Double-Q learning*, as TD3 learns two Q-functions instead of one, $Q_{\phi 1}$ and $Q_{\phi 2}$. From the two Q-values, the smaller one is used to form the targets in the Bellman error loss functions. The learning itself is done as in DDPG with the minimization of MSBE. The target update for the Clipped Double Q-learning algorithm is defined as [5]:

$$y = r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \pi_\phi(s')) \tag{5.8}$$

This method cannot lead to additional overestimation over using the standard Q-learning target [5]. However, it may introduce an underestimation bias. Unlike overestimation, it does not create problems as the value of underestimated actions will not be propagated through the policy update [5].

Another important upgrade is delaying the update of the policy network. Authors of [5], based on the experiments, conclude with the following remark: “If target networks can be used to reduce the error over multiple updates, and policy updates on high-error states cause divergent behavior, then the policy network should be updated at a lower frequency than the value network, to first minimize error before introducing a policy update.” The update is in the form of:

$$\theta' \leftarrow \tau \cdot \theta + (1 - \tau)\theta' \tag{5.9}$$

The policy and target networks are updated only after a fixed number of d critic updates. The likelihood of repeating updates with respect to an unchanged critic is limited by sufficiently delaying the policy updates. This way, the computed policy updates will use a value estimate with a lower variance, which should lead to higher-quality policy updates [5].

Another way of reducing the variance is using the *Target Policy Smoothing Regularization*. The training procedure is modified to explicitly force similar actions to have similar values. In [5], they propose to fit the value of a small area around the target action to

$$y = r + \mathop{\mathbb{E}}_{\epsilon}[Q_{\theta'}(s', \pi_{phi'}(s') + \epsilon)]. \tag{5.10}$$

This change allows for smoothing the value estimate by calculating it from similar state-action value estimates. In practice, the expectation over actions can be approximated by averaging over mini-batches where a small amount of noise is added to the target.

$$\begin{aligned} y =& r + \gamma Q_{\theta'}(s', \pi_{\phi'}(s') + \epsilon), \\ \epsilon \sim& \text{clip}(\mathcal{N}(0, \sigma), -c, c), \end{aligned} \tag{5.11}$$

where σ stands for a noisy policy. The added noise is clipped to keep the target close to the original action. This leads to safer policies and can additionally lead to improvement in stochastic domains with failure cases.

5.1.3 Implementation

As this algorithm is used in this thesis, the implementation from the Stable Baselines 3 [42] library is utilized. It is the base implementation of the TD3 algorithm with all hyperparameters adjustable. Pytorch [43] library is used in the Stable Baselines 3 to implement the networks.

5.2 Model-based methods

Model-based methods learn the model of the environment in addition to the agent's policy. As stated in Section 4.5.2, they are more complex and computationally more expensive as more calculations need to be done per step of the algorithm. The reward signal from Section 5.3 is used.

5.2.1 Dreamer

Dreamer [6] is a model-based algorithm that utilizes *latent imagination* to increase the sample learning efficiency. The latent imagination process simulates the experience in the learned latent space (Section 4.5.7) with the learned model. The agent learns long-horizon behaviors from observations purely by latent imagination. An actor-critic algorithm is introduced to account for rewards beyond the imagination horizon while utilizing neural network dynamics. For this, the agent predicts state values and actions in the learned latent space. The values are used to optimize the Bellman consistency for imagined rewards, while the policy maximizes the values by propagating their analytic gradients back through the dynamics.

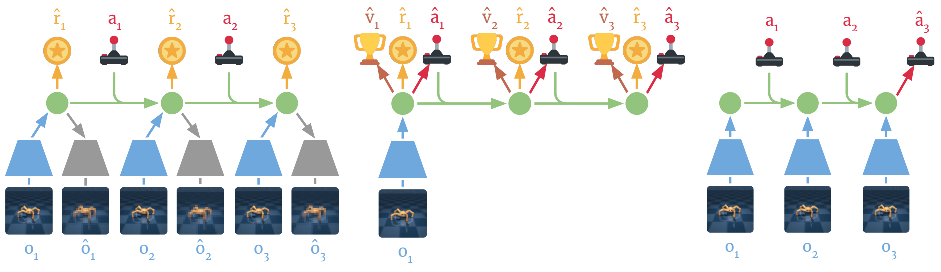
Problem definition

The problem is formulated as a POMDP with a discrete time step $t \in [1, T]$. Action space is continuous with action a_t generated from the agent, $a_t \sim p(a_t|o_{\leq t}, a_{<t}$, and observation space is high dimensional. The rewards are scalar, generated with the observation of an unknown environment. Both the observations o_t, and the rewards r_t are generated by the unknown environment, $o_t, r_t \sim p(o_t, r_t|o_{<t}, a_{<t})$. As usual in RL, the goal is to maximize the expected sum of rewards $\mathbb{E}\left(\sum_{t=1}^{T} r_t\right)$.

A *latent dynamics* model is used. It consists of 3 parts:

$$\begin{aligned}
&\text{Representation model: } p(s_t|s_{t-1}, a_{t-1}, o_t),\\
&\text{Transition model: } q(s_t|s_{t-1}, a_{t-1}),\\
&\text{Reward model: } q(r_t|s_t),
\end{aligned} \tag{5.12}$$

where p is the distribution that generates samples in the real environment, and q is the approximation used to predict samples in the latent imagination. Observations and actions are encoded to create a vector of values for the model states s_t with Markovian transitions [6]. The reward model predicts rewards given the model states, and the transition model predicts future model states without seeing the corresponding observations. This allows predictions in the compact latent space by the transition model without the need to interact with the environment and its high-dimensional observations.


$\hat{r}_1$ a_1 $\hat{r}_2$ a_2 $\hat{r}_3$
o_1 $\hat{o}_1$ o_2 $\hat{o}_2$ o_3 $\hat{o}_3$
$\hat{v}_1$ $\hat{r}_1$ $\hat{a}_1$ $\hat{v}_2$ $\hat{r}_2$ $\hat{a}_2$ $\hat{v}_3$ $\hat{r}_3$ $\hat{a}_3$
o_1
a_1 a_2 $\hat{a}_3$
o_1 o_2 o_3

(a) : Learn dynamics from experience **(b) :** Learn behavior in imagination **(c) :** Act in the environment

Figure 5.1: Components of Dreamer. (a) From the dataset of past experience, the agent learns to encode observations and actions into compact latent states (●), for example via reconstruction, and predicts environment rewards (★). (b) In the compact latent space, Dreamer predicts state values (🏆) and actions (🕹) that maximize future value predictions by propagating gradients back through imagined trajectories. (c) The agent encodes the history of the episode to compute the current model state and predict the next action to execute in the environment. Figure taken from [6]

The RL agent's components are dynamics learning, behavior learning, and environment interaction. Predicting the hypothetical trajectories in the compact latent space world model is used to learn the behavior. The sequence of actions in a learning episode can be seen in Fig. 5.1. Depending on the problem and implementation, some of the three steps may be done in parallel. The implementation from [17] used in this thesis is running these steps in sequence.

Behavioral learning

The behavior is trained in the compact latent space of the learned world model.

Imaginary environment. In the imaginary environment, the time step is denoted as τ to differentiate between it and a timestep t in the environment. Imagined trajectories start at the true model states s_t, drawn from the agent's experience combined with the observations. These trajectories then follow the predictions of (i) the transition model: $s_\tau \sim q(s_\tau|s_{\tau-1}, a_{\tau-1})$, (ii) the reward model: $r_\tau \sim q(r_\tau|s_\tau)$, and (iii) the policy: $a_\tau \sim q(a_\tau|s_\tau)$. The objective is to maximize the expected rewards with respect to the policy in an imaginary environment:

$$\max \mathop{\mathbb{E}}_{q} \Big(\sum_{\tau=t}^{\infty} \gamma^{\tau-t} r_\tau \Big). \tag{5.13}$$

Action and value models. An actor-critic approach is used to learn behaviors with regard to rewards beyond a horizon H. Action and value models are learned in the latent space of the world model. The action model predicts policy, and thus action, $a_\tau \sim q_\phi(a_\tau|s(\tau))$, for the imaginary environment, while the value model, $v_\psi(s_\tau) \approx \mathbb{E}_{q(\cdot|s_\tau)}\left(\sum_{\tau=t}^{t+H}\gamma^{\tau-t}r_\tau\right)$, estimates the expected imaginary rewards for the predicted actions for each state s_τ.

For both action and value models, dense neural networks are used. The value model network has parameters ψ. The action model network with parameters ϕ outputs a tanh-transformed Gaussian. The sufficient statistics are predicted by the neural network [6]. This enables reparameterized sampling, which treats the sampled actions as deterministically dependent on the output of the neural network. Consequently, that allows for backpropagation of analytic gradients through the sampling process,

$$a_\tau = \tanh(\mu_{phi}(s_\tau) + \sigma_\phi(s_\tau)\epsilon), \quad \epsilon \sim \mathrm{Normal}(0, \mathbb{I}). \tag{5.14}$$

Value estimation. Value of the imagined trajectories $\{s_\tau, a_\tau, r_\tau\}_{\tau=t}^{t+H}$ have to be estimated in order to learn the action and value models. These trajectories are developed from the world states s_t, which are sequence batches drawn from the agent's experience dataset. Using actions sampled from the action model, the trajectories are used to predict forward for the imagination horizon H. The state values can be estimated in multiple ways. The one used in Dreamer is:

$$\mathrm{V}_\lambda(s_\tau) \approx (1-\lambda)\sum_{n=1}^{H-1}(\lambda^{n-1}\mathrm{V}_\mathrm{N}^n(s_\tau)) + \lambda^{H-1}\mathrm{V}_\mathrm{N}^H(s_\tau), \tag{5.15}$$

where V_λ is an exponentially weighted average of the estimates for different k used to balance bias and variance.

Learning objective. The value estimates $\mathrm{V}_\lambda(s_\tau)\forall s_\tau$ along the imagined trajectories have to be computed in order to update the action and value models. The action model $q_\phi(a_\tau|s_\tau)$ has the objective of maximizing the value estimates by predicting actions and state trajectories:

$$\max_\phi \mathop{\mathbb{E}}_{q_\theta, q_\phi}\left(\sum_{\tau=t}^{t+H}\mathrm{V}_\lambda(s_\tau)\right). \tag{5.16}$$

The objective of the value model $v_\psi(s_\tau)$, in turn, is to regress the value estimates [34]:

$$\min_\psi \mathop{\mathbb{E}}_{q_\theta, q_\phi}\left(\sum_{\tau=t}^{t+H}\frac{1}{2}\left\|v_\psi(s_\tau) - \mathrm{V}_\lambda(s_\tau))\right\|^2\right). \tag{5.17}$$

The value model is updated to regress the targets around which the gradient is stopped [34]. The action model uses analytic gradients through the learned dynamics to maximize the value estimates. As the action model is dependent on the value estimates, the estimates depend on the value and reward predictions. These predictions are further dependent on imagined states, and those are dependent and given by the imaginary actions from the model. This chain allows the gradient of the sum of the expectations:

$$\nabla_\phi \mathop{\mathbb{E}}_{q_\theta, q_\phi} \Big(\sum_{\tau=t}^{t+H} \mathrm{V}_\lambda(s_\tau) \Big), \tag{5.18}$$

to be analytically computed by stochastic backpropagation. The world model and its parameters are fixed while learning the behaviors. Reparametrization is used for continuous actions. The latent states and straight-through gradients are used for discrete actions.

Latent dynamics

Latent dynamics are dynamics that apply in the latent space. They are used for simulation in the latent imagination. Three ways to learn latent dynamics are presented in the paper that introduces Dreamer [6]. We described only the *Reconstruction*, as it is used in the implementation from [17].

Reconstruction. The world model includes the following components,

$$\begin{aligned}
&\text{Representation model:} && p_\theta(s_t|s_{t-1}, a_{t-1}, o_t) \\
&\text{Observation model:} && q_\theta(o_t|s_t) \\
&\text{Reward model:} && q_\theta(r_t|s_t) \\
&\text{Transition model:} && q_\theta(s_t|s_{t-1}, a_{t-1}).
\end{aligned} \tag{5.19}$$

The observation model is only utilized as a learning signal. All of the models are optimized jointly rather than in a sequence. This is done in order to increase the variational lower bound:

$$\begin{aligned}
&\mathcal{J}_{\mathrm{REC}} \doteq \mathop{\mathbb{E}}_{p} \Big(\sum_t \big(\mathcal{J}_{\mathrm{O}}^t + \mathcal{J}_{\mathrm{R}}^t + \mathcal{J}_{\mathrm{D}}^t \big) \Big) + \text{const} \qquad \mathcal{J}_{\mathrm{O}}^t \doteq \ \ln q(o_t|s_t) \\
&\mathcal{J}_{\mathrm{R}}^t \doteq \ln q(r_t|s_t) \qquad \mathcal{J}_{\mathrm{D}}^t \doteq \ -\beta \mathrm{KL}(p(s_t|s_{t-1}, a_{t-1}, o_t))q(s_t|s_{t-1}, a_{t-1})
\end{aligned} \tag{5.20}$$

The variational lover bound, J_{REC}, includes reconstruction terms for observations $\mathcal{J}_{\mathrm{O}}^t$, rewards $\mathcal{J}_{\mathrm{R}}^t$, and a *Kullback–Leibler* (KL) regularizer $\mathcal{J}_{\mathrm{D}}^t$. While in the original paper, the representation model is implemented as a combination of a Convolutional Neural Network (CNN) and a Recurrent State Space Model (RSSM), in [17], it is implemented as a Multi Layer Perceptron (MLP). The observation model is implemented as a transposed CNN, the reward model as a dense network, and the transition model as an RSSM. The combined parameter vector θ is updated by stochastic backpropagation [6].

Model components		**Hyper parameters**	
Representation	$p_\theta(s_t|s_{t\text{-}1}, a_{t\text{-}1}, o_t)$	Seed episodes	S
Transition	$q_\theta(s_t|s_{t\text{-}1}, a_{t\text{-}1})$	Collect interval	C
Reward	$q_\theta(r_t|s_t)$	Batch size	B
Action	$q_\phi(a_t|s_t)$	Sequence length	L
Value	$v_\psi(s_t)$	Imagination horizon	H
		Learning rate	α

Algorithm 5: Dreamer [6]

```
Initialize dataset $\mathcal{D}$ with $S$ random seed episodes.
Initialize neural network parameters $\theta, \phi, \psi$ randomly.
while not converged do
    for update step $c = 1..C$ do
        // Dynamics learning
        Draw $B$ data sequences $\{(a_t, o_t, r_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
        Compute model states $s_t \sim p_\theta(s_t|s_{t-1}, a_{t-1}, o_t)$.
        Update $\theta$ using representation learning.
        // Behavior learning
        Imagine trajectories $\{(s_\tau, a_\tau)\}_{\tau=t}^{t+H}$ from each $s_t$.
        Predict rewards $\mathbb{E}(q_\theta(r_\tau|s_\tau))$ and values $v_\psi(s_\tau)$.
        Compute value estimates $\mathrm{V}_\lambda(s_\tau)$ via .
        Update $\phi \leftarrow \phi + \alpha\nabla_\phi \sum_{\tau=t}^{t+H} \mathrm{V}_\lambda(s_\tau)$.
        Update $\psi \leftarrow \psi - \alpha\nabla_\psi \sum_{\tau=t}^{t+H} \frac{1}{2}\left\|v_\psi(s_\tau) - \mathrm{V}_\lambda(s_\tau)\right\|^2$.
    end
    // Environment interaction
    $o_1 \leftarrow$ env.reset()
    for time step $t = 1..T$ do
        Compute $s_t \sim p_\theta(s_t|s_{t-1}, a_{t-1}, o_t)$ from history.
        Compute $a_t \sim q_\phi(a_t|s_t)$ with the action model.
        Add exploration noise to action.
        $r_t, o_{t+1} \leftarrow$ env.step($a_t$).
    end
    Add experience to dataset $\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t, r_t)_{t=1}^{T}\}$.
end
```

Implementation

The implementation from [17] is selected to be used in this work. It was chosen because it has already successfully run on the F1TENTH platform. The implementation is modified to be used with the ROS2 on our build of the car. The reward signal described in Section 5.3 is further used. An effort to implement Dreamer v2 [18] has been made in this work. However, the new object design changes were too big to implement without refactoring the whole code.

5.3 Reward signal

We present a reward signal in addition to the base reward of progress made in a single time step used in [17]:

$$r_{progress} = (cp(s_t) - cp(s_{t-1})), \tag{5.21}$$

where $cp(s_t)$ is a projection on the central line in time step t. The difference between the projections is the progress made in a single time step. We added a small constant negative reward in each time step to promote optimization of the lap time. The progress reward will be the same as the agent drives faster lap times, but fewer penalties will be accrued.

As mostly the TD3 agent often got stuck in training and refused to move, a large constant as a penalty for standing still multiple time steps was added. This helped the agent start moving at the beginning of the learning. As the agent learned to progress through the tracks, it did not stop, so this part of the reward did not impact the later stages of the learning.

Chapter 6

Experiments

In this section, we introduce and describe the tracks selected for the experiments. On each track, we train a TD3 [5] agent and a Dreamer [6] agent. Agents are tested on all tracks, not only the one they were trained on. Results are then provided in Chapter 7.

6.1 Columbia

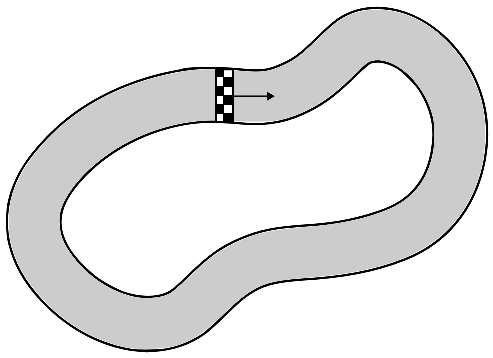


Figure 6.1: Columbia track.

The track is 61.20 m long and constantly 3.55 m wide. This track consists of long and wide turns. Also, it is the only track where the agent drives in a clockwise direction. This means a lack of left-hand turns that are dominant in the rest of the experiments. It was chosen to showcase the agent's behavior, which was trained in a simple environment and then tested in more complex ones. Also, with the modified reward function (Section 5.3), optimization of the lap time is observed.

6.2 Austria

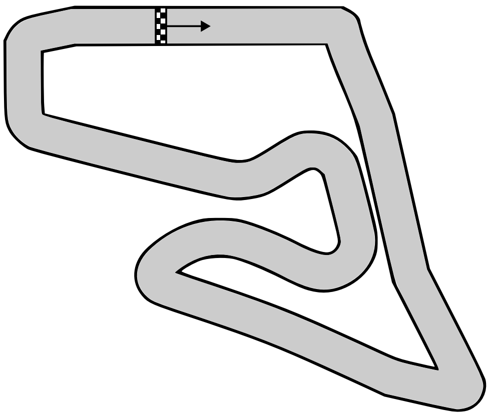


Figure 6.2: Austria track.

A scaled-down version of a popular F1 track based in Austria. The track is 79.45 m long, and the straights are 1.9 m wide. This track was chosen as a more complex problem to solve for the agent. It consists of two straights where high speed can be achieved, followed by a sequence of sharp turns where slower speeds are needed to pass through safely.

The straights are longer than the lidar range, so it is challenging to control the speed accordingly. If the agent is going too fast, at the moment it detects the end of the straight, it will not be able to slow down enough for the turn.

The sequence of turns is to show the behavior in a complex sequence of turns where the End-to-End agent does not see what is coming next. An Additional TD3 agent from learning that did run for a shorter time is presented.

6.3 Treitlstrasse

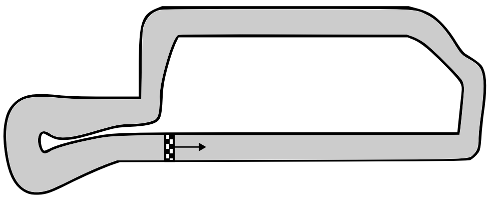


Figure 6.3: Treitlstrasse track [17].

This is the map used for the demonstration in [17]. The track is 51.65 m long, 1.35 m wide at the starting line, and 0.90 m in the narrowest point. This track was chosen to observe the agents behave in narrow corridors. This track was the real-world track where the authors from [17] successfully tested the Dreamer agent.

6.4 CIIRC

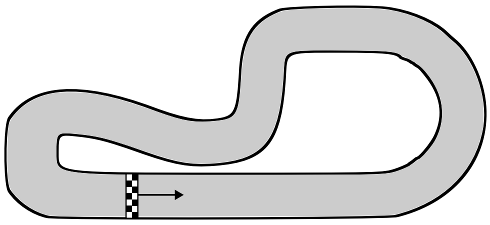


Figure 6.4: CIIRC track.

This track is used in real-life experiments. The track consists of a 15 m long straight followed by a wide 180° left-hand turn, a chicane that continues into a tight left-hand hairpin. It is 38.20 m long and constantly 1.9 m wide. It is the shortest one because it is within the restrictions of our environment. It was designed to test the agent's capability in multiple different turns and situations. Two TD3 agents with different network architectures and hyperparameters are trained and tested along with one Dreamer agent.

Chapter 7

Results

In this section, we describe the results of all of the experiments. We provide a baseline of the Follow the Gap algorithm, a purely reactive algorithm described in Section 4.1.3. This algorithm was chosen for a baseline because it does not utilize any more information than what we use in the RL agents.

We show both agents' training process and evaluation results on each track. Each agent was trained for 50 hours on a cluster with the same hardware.

In the training graph, e.g., Fig. 7.3, the y axis shows the progress on the training track, where 1 signals a completed lap. The x axis shows the number of time steps, where one time step is $0.01\,\mathrm{s}$ in simulation.

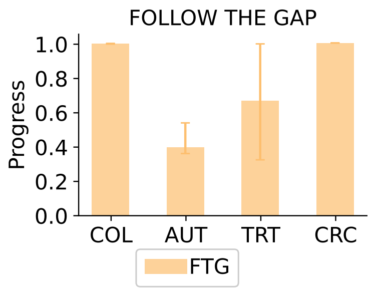
The agents were evaluated on each track, not only the one they were trained on. Each of these evaluations ran five times. We provide two bar charts and all of the trajectories with the results.

In the first evaluation bar chart, e.g., Fig. 7.1a, we show the mean of progress on each track. The mean lap time is shown in the second bar chart, e.g., Fig. 7.1b. The delimiters on both bar charts show the minimum and maximum of the shown values.

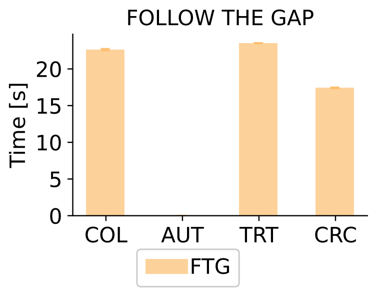
The agents' trajectories, e.g., Fig. 7.2, are shown with the velocities at each point. On each figure, all of the five trajectories from the evaluations are shown. The color of the trajectory indicates the velocity. The color bar on the right of each trajectory describes each color's value.

7.1 Baseline: Follow the Gap algorithm

Here, we can see the baseline to compare the rest of the algorithms. As we can see in Fig. 7.1, the Follow the Gap algorithm did not manage to complete the Austria track, Fig. 7.2b as it crashed in sharp turns. It could not keep going in a straight line on the straights, and it needed to adjust frequently, which resulted in a wavy trajectory. This pattern can be seen on all tracks except the widest one, Columbia. In some runs on Treitlstrasse, Fig. 7.2c, the FTG crashed. One run crashed in the first corner, two runs crashed in the tight chicane, and two managed to complete a lap. The performance on the CIIRC track, Fig. 7.2d, and the Columbia track, Fig. 7.2a, was stable as all the runs behaved consistently and provided the same results.

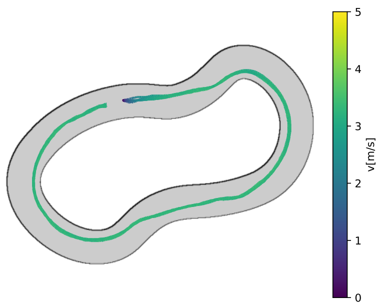


(a) : The progress in evaluation.

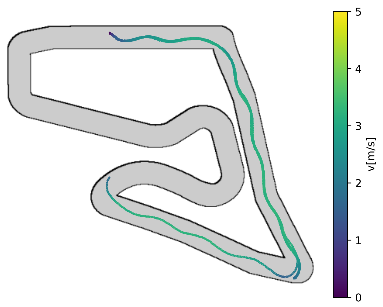


(b) : Lap times achieved.

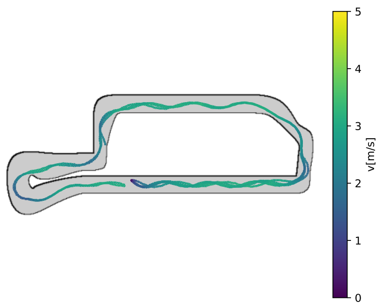
Figure 7.1: Evaluation of the Follow the Gap agent.



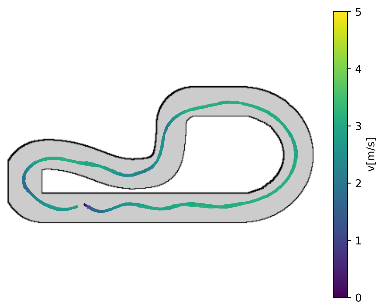
(a) : The trajectory on Columbia.



(b) : The trajectory on Austria.



(c) : The trajectory on Treitlstrasse.



(d) : The trajectory on CIIRC.

Figure 7.2: The trajectories of the Follow the Gap agent.

7.2 Columbia

7.2.1 Training

Both agents completed multiple laps in the training, Fig. 7.3, on the Columbia track. The TD3 agent completed four laps in the training before an issue happened, and the agent stopped learning and crashed to 0 reward. We suppose it is a software problem. This problem was reoccurring. The Dreamer agent only managed to complete two laps.

Also, in Fig. 7.3, we can see that although both agents had the same resources and time to learn, the TD3 agent completed marginally more time steps. This holds true for all of the experiments.

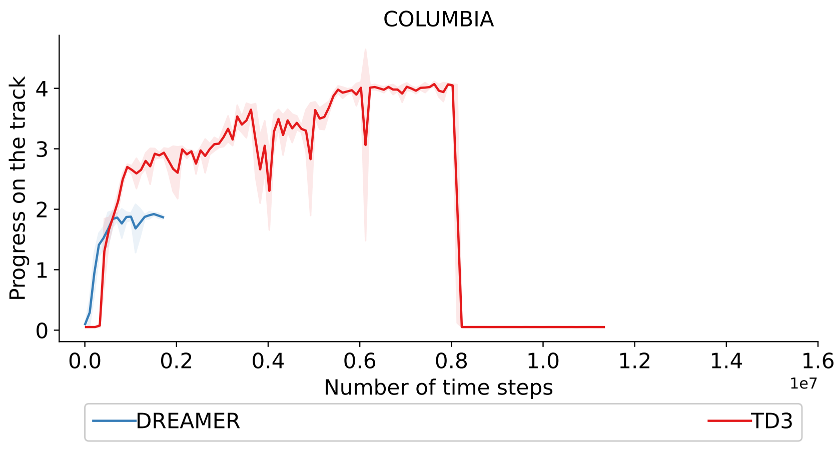
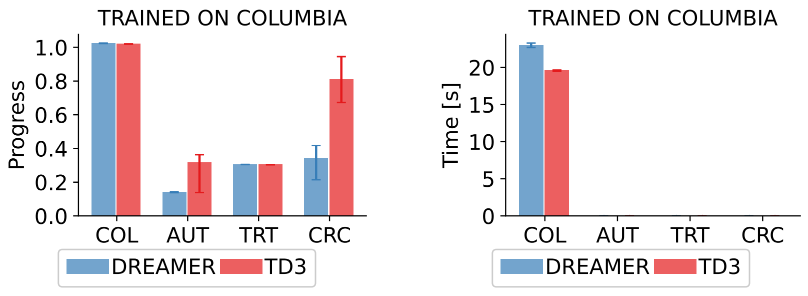


Figure 7.3: The course of the training for the Columbia agent.

7.2.2 Evaluation

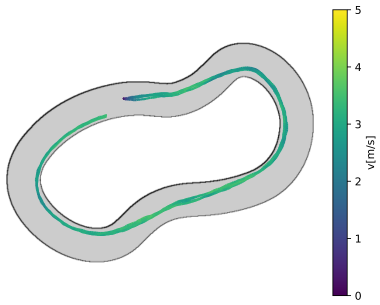
The Columbia track is the widest, with wide turns only. This means it is the least complex for the agents to complete. As seen in the training, the TD3 agent excelled on this track and even managed to be consistently faster than the FTG baseline. The Dreamer agent successfully managed to complete all testing laps.



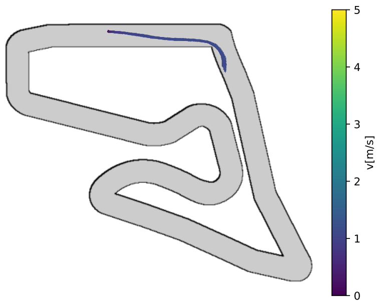
(a) : The progress in evaluation. **(b) :** Lap times achieved.

Figure 7.4: Evaluation of the Columbia agent.

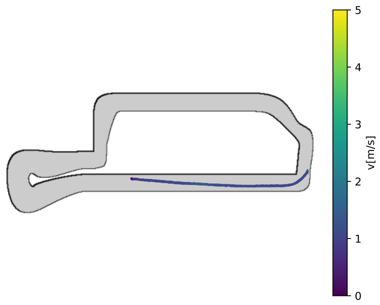
Neither of the agents managed to finish laps on any other track than the one they had been trained on. The progress both agents made is shown in Fig. 7.4a. The Dreamer agent could not handle more narrow tracks, as he moved about a meter per second on other tracks, as can be seen in Fig. 7.5. While the TD3 agent, shown in Fig. 7.6, managed to drive at higher speeds and almost completed the CIIRC track, Fig. 7.6d, where he did not manage the last hairpin.



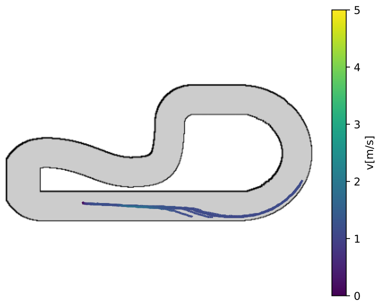
(a) : The trajectory on Columbia.



(b) : The trajectory on Austria.

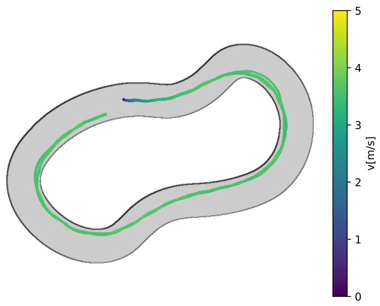


(c) : The trajectory on Treitlstrasse.

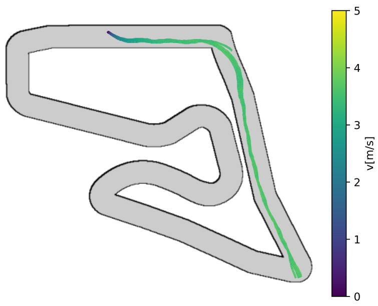


(d) : The trajectory on CIIRC.

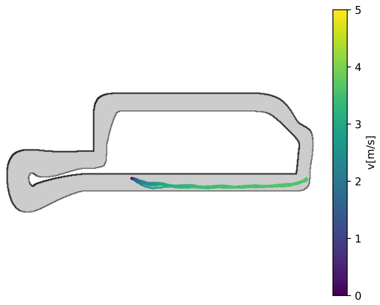
Figure 7.5: The trajectories of the Columbia Dreamer agent.



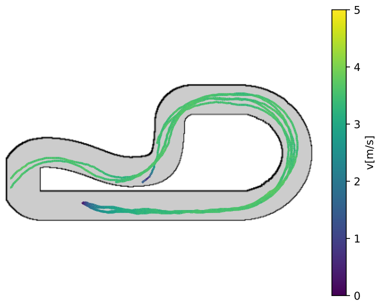
(a) : The trajectory on Columbia.



(b) : The trajectory on Austria.



(c) : The trajectory on Treitlstrasse.



(d) : The trajectory on CIIRC.

Figure 7.6: The trajectories of the Columbia TD3 agent.

7.3 Austria

7.3.1 Training

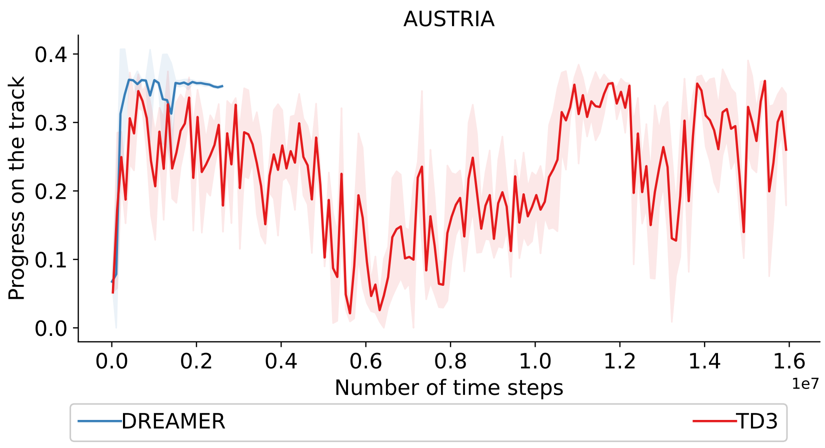
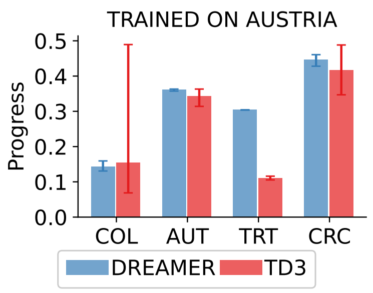


Figure 7.7: The course of the training for the Austria agent.

In Fig. 7.7, we can see that the TD3 learning process is less stable and often gets much lower rewards than it already managed to get, while the Dreamer agent does not have such reward drops. Neither agent managed a full lap in the training.

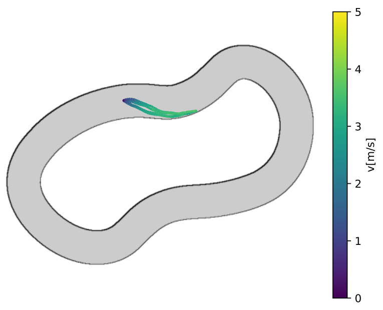
7.3.2 Evaluation

As no full laps were completed by either of the agents, only a bar chart with the progress is shown Fig. 7.8. Interestingly, the learned agents performed the best on the CIIRC track, not the Austria one.

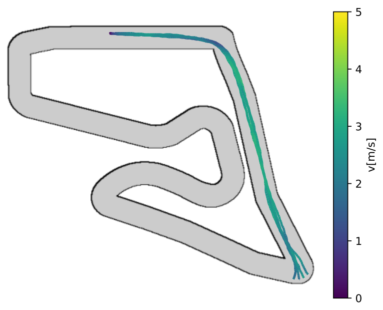


(a) : The progress in evaluation.

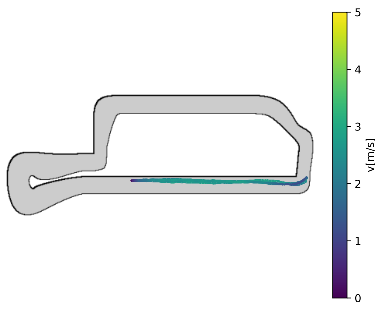
Figure 7.8: Evaluation of the Austria agent.



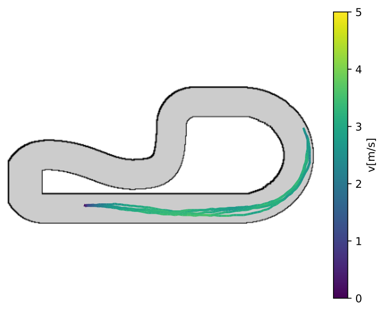
(a) : The trajectory on Columbia.



(b) : The trajectory on Austria.

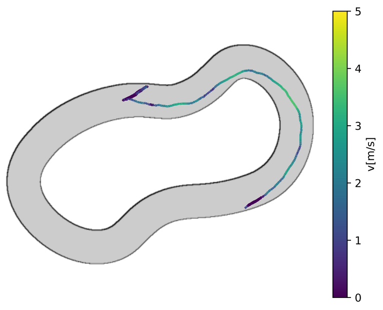


(c) : The trajectory on Treitlstrasse.

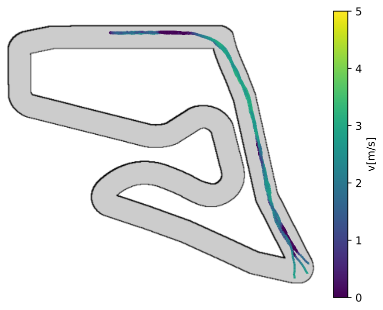


(d) : The trajectory on CIIRC.

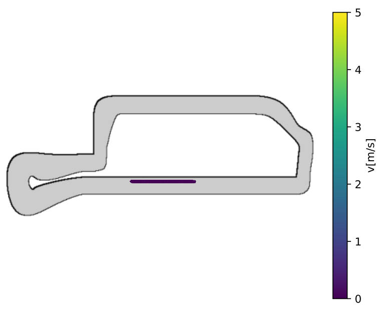
Figure 7.9: The trajectories of the Austria Dreamer agent.



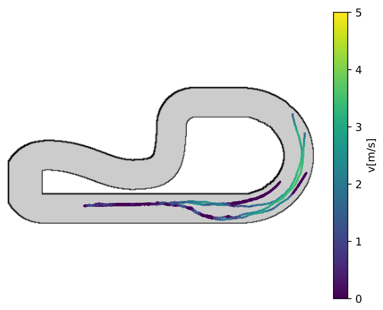
(a) : The trajectory on Columbia.



(b) : The trajectory on Austria.



(c) : The trajectory on Treitlstrasse.



(d) : The trajectory on CIIRC.

Figure 7.10: The trajectories of the Austria TD3 agent.

As can be seen in both Fig. 7.9b and Fig. 7.10b, the agents are not able to get past the second turn. The turn is sharp, and as seen in Fig. 7.2b, not even the baseline algorithm of FTG could pass this turn reliably. Apart from both agents making progress into the first turn on the CIIRC track, Figs. 7.9d and 7.10d, the TD3 agent managed once almost to complete half of the Columbia track as shown in Fig. 7.10a. In Fig. 7.10d, we can see that even though sometimes, the agent progresses further than even the Dreamer agent in Fig. 7.9d, the behavior is less consistent as the trajectories vary a lot.

7.4 Treitlstrasse

7.4.1 Training

Here, we also present the TD3 agent that encountered a crash of the cluster during the training. As seen in Fig. 7.11, one TD3 agent finished the whole training. However, it encountered the same issue as in the Columbia training. The agent got stuck and did not progress. On the other hand, the one that crashed showed some progress and was used in the evaluation part.

As shown in Fig. 7.11, the learning process of the Dreamer agent was more volatile and had some reward drops.

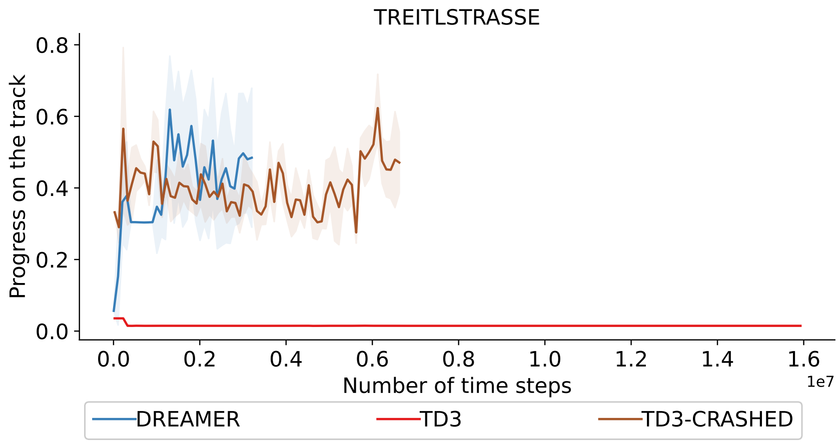
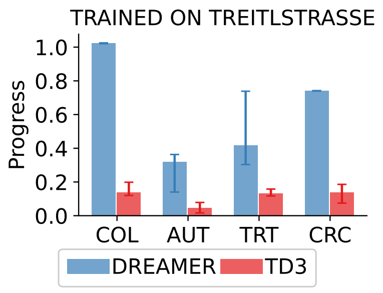


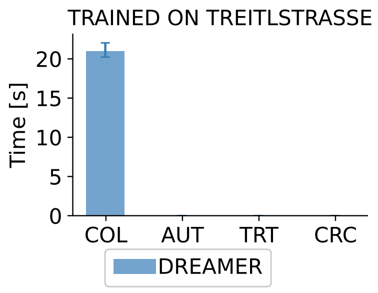
Figure 7.11: The course of the training for the Tretlstrasse agent.

7.4.2 Evaluation

In Fig. 7.12, we see that the Dreamer agent was the only agent who managed to complete a lap on any track. It completed all five laps on the Columbia track and even got better times than the Columbia Dreamer or baseline FTG agent. The TD3 almost did not move in the evaluations, as its speed was around one meter per second.



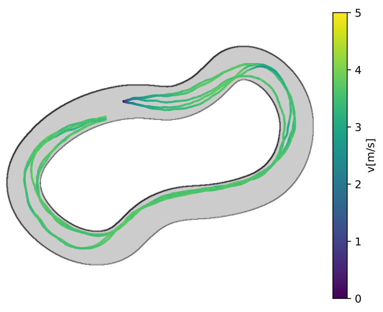
(a) : The progress in evaluation.



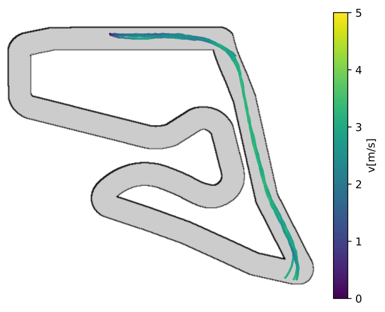
(b) : Lap times achieved.

Figure 7.12: Evaluation of the Treitlstrasse agent.

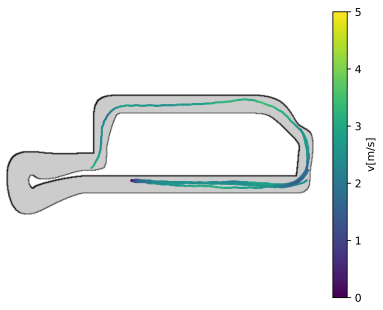
As we see in Figs. 7.13c and 7.13d, the agent in both cases progressed well until the chicane, where it crashed at the exit. As shown in Fig. 7.14, the TD3 agent made no noticeable progress.



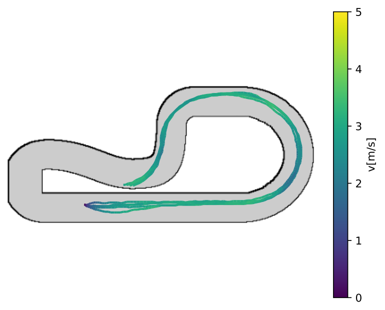
(a) : The trajectory on Columbia.



(b) : The trajectory on Austria.

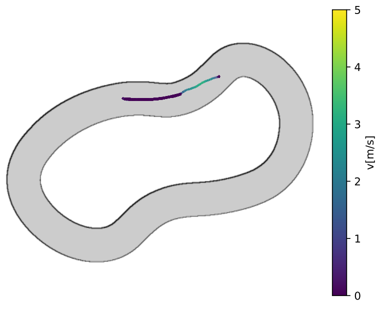


(c) : The trajectory on Treitlstrasse.

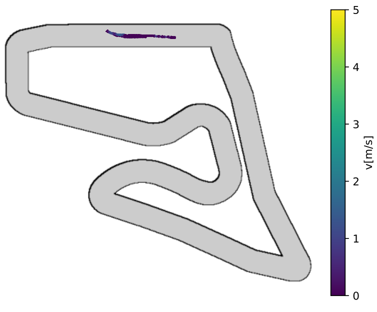


(d) : The trajectory on CIIRC.

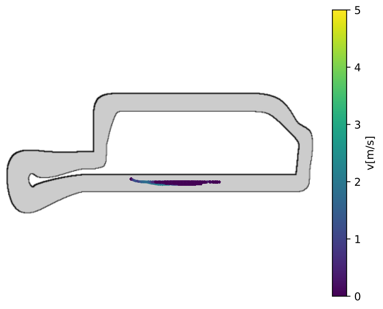
Figure 7.13: The trajectories of the Treitlstrasse Dreamer agent.



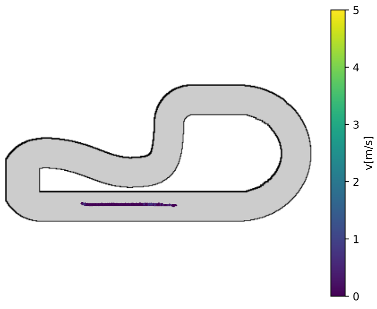
(a) : The trajectory on Columbia.



(b) : The trajectory on Austria.



(c) : The trajectory on Treitlstrasse.



(d) : The trajectory on CIIRC.

Figure 7.14: The trajectories of the Treitlstrasse TD3 agent.

7.5 CIIRC

7.5.1 Training

In this case, Fig. 7.15, the Dreamer algorithm clearly outperformed the TD3 one in the training. The TD3 agent had more stable rewards during the training. The Dreamer agent managed to complete two laps while learning.

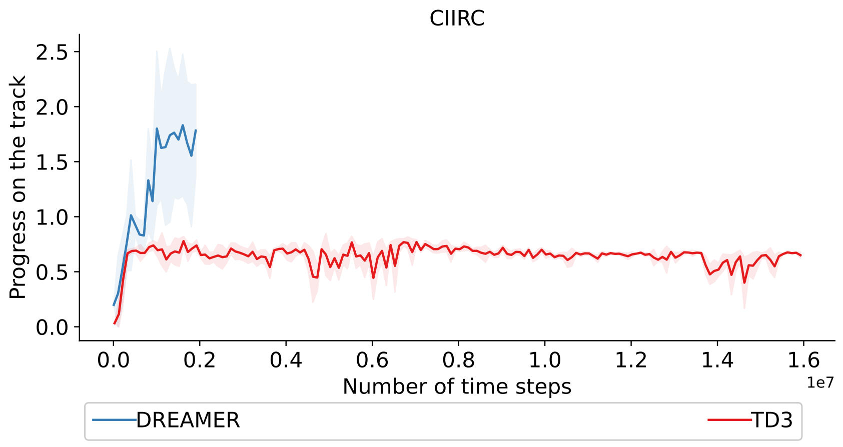
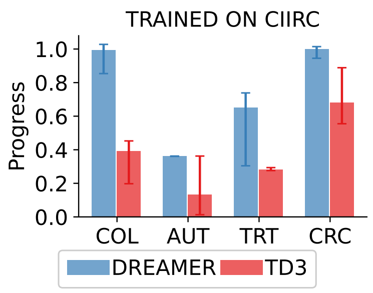


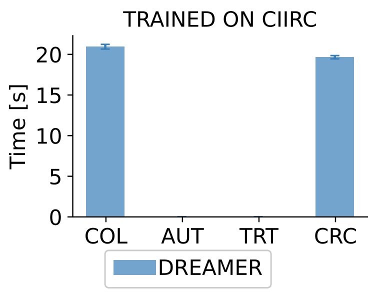
Figure 7.15: The course of the training for the CIIRC agent.

7.5.2 Evaluation

The Dreamer CIIRC agent is the only agent who successfully completed a lap on a track other than the Columbia. The time on Columbia was slightly faster than the baseline FTG agent, while the time on the CIIRC track was slower than the baseline.

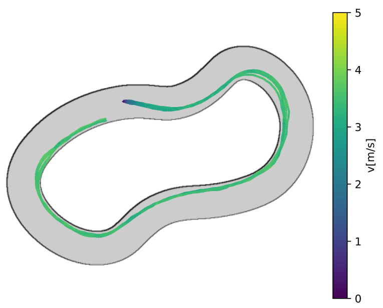


(a) : The progress in evaluation.

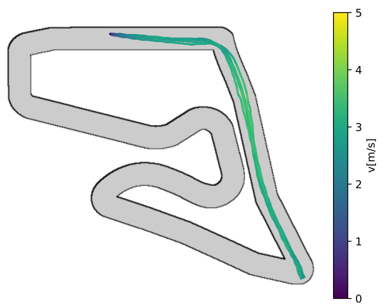


(b) : Lap times achieved.

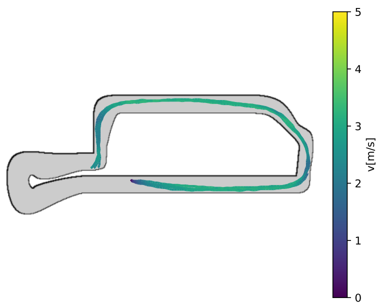
Figure 7.16: Evaluation of the CIIRC agent.



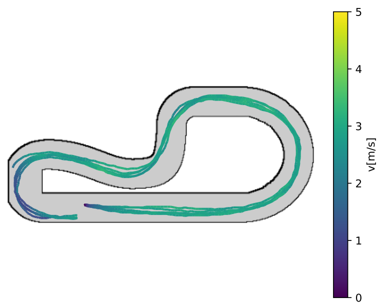
(a) : The trajectory on Columbia.



(b) : The trajectory on Austria.

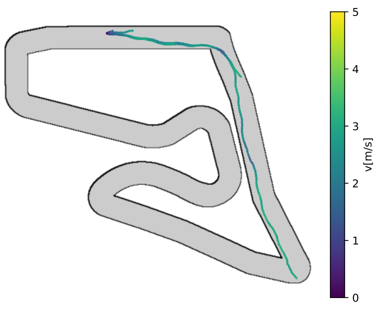


(c) : The trajectory on Treitlstrasse.

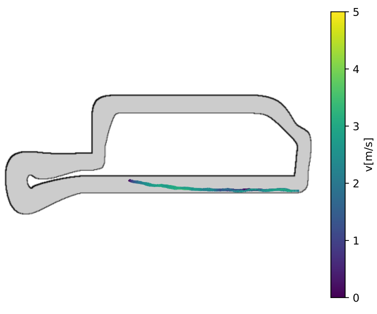


(d) : The trajectory on CIIRC.

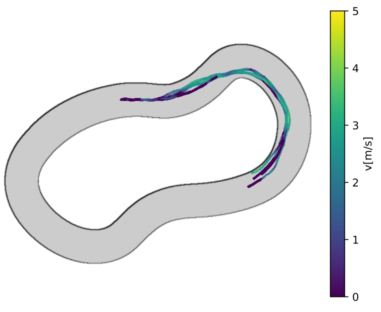
Figure 7.17: The trajectories of the CIIRC Dreamer agent.



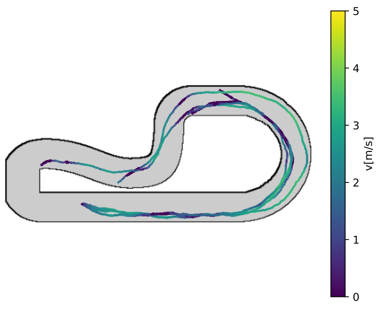
(a) : The trajectory on Austria.



(b) : The trajectory on Treitlstrasse.



(c) : The trajectory on Columbia.



(d) : The trajectory on CIIRC.

Figure 7.18: The trajectories of the CIIRC TD3 agent.

The Dreamer agent on the CIIRC track, Fig. 7.17d, crashed two times in the last hairpin turn, but all trajectories are consistent and nearly identical. Also, on the Treitlestrasse track, Fig. 7.17c, we can see that the width of the chicane can be the difference between crashing and driving through safely. The chicane is visibly more narrow than the one on the CIIRC track. The TD3 agent, Fig. 7.18d, progressed through the track and crashed at the beginning of the last hairpin turn. However, the TD3 agent is less consistent as most runs crashed earlier in the lap.

7.6 Real-life experiments on CIIRC track

We successfully made both TD3 and Draemer agents run on our F1TENTH platform. In this experiment, the minimum speed is set to $1\,\mathrm{m}\cdot\mathrm{s}^{-1}$ and the maximum is $3\,\mathrm{m}\cdot\mathrm{s}^{-1}$. The minimum is set because any lower speed than that and the motor cannot move the car. The maximum is set because of safety and the difference in the traction between the simulation and the real world. The agents were briefly tested with higher limits and did not have enough grip to steer or brake in time. The lap times were measured using an optical gate.

Only the agents trained on the CIIRC track and the FTG agent managed to finish the track. The FTG agent used was different than the one used in simulations. Each agent was given three tries to finish the track. Both D3 and Dreamer agents trained on the CIIRC track crashed twice in the last hairpin turn.

The agent	Lap time
FTG	23.95 s
TD3	22.54 s
Dreamer	22.35 s

Table 7.1: Lap time from the real-life experiments.

7.6.1 Dreamer agents

The Dreamer agent's trajectory in real life was not as smooth as in simulation. The agent needed to steer more often to stay within the walls, and its trajectory was similar to the one from FTG in simulation (Fig. 7.2d). We argue that this is caused by the sim-to-real gap as the model in the simulation is not that accurate since we do not possess the measurements needed to set up everything. The traction in the simulator was set too high compared to the actual traction on the floor the agents drove on. This was set during the modeling of the track for the simulator, and the friction of the surface was set as in [17]. Their experiment, like ours, was conducted indoors on a linoleum floor.

The Columbia Dreamer agent could not stay within the walls on the straight at the start and did not progress even to the first corner. The Dreamer agents trained on Austria and Treitlstrasse tracks behaved closely to the simulation. The Austria agent crashed constantly at the end of the first corner, while the Treitlstrasse agent could not drive through the chicane successfully.

7.6.2 TD3 agents

The TD3 agents ran into the same problem as happened in training in Figs. 7.3 and 7.11. Sometimes, they only returned full steer and full throttle. This resulted in an instantaneous crash. Only the CIIRC and Treitlstrasse (with incomplete learning) agents provided meaningful results. The CIIRC agent, as seen in Table 7.1, successfully completed a lap. Like the CIIRC Dreamer agent, the CIIRC TD3 agent crashed twice in the last hairpin turn. The pace the CIIRC agent drove was faster than in the simulation. We argue that this results from setting the minimum speed, which helped the agent to move.

7.7 Discussion

We have successfully implemented, trained, and evaluated two Reinforcement Learning agents. Both algorithms, TD3 and Dreamer, finished a lap in a real-world scenario. Both algorithms used a reward signal defined in Section 5.3 to promote faster driving while not compromising safety.

7.7.1 TD3 algorithm

The implementation from Stable Baselines 3 [42] did not work properly, and in both training and evaluation, it often returned actions that led to crashes.

Changing hyperparameters did not have an effect on this problem. We suspect it ran into some software-related issues. We did not provide data from many training attempts with the same result as the TD3 agent in Fig. 7.11.

Apart from those issues, the only agent that consistently drove faster than $2\,\mathrm{m}\cdot\mathrm{s}^{-1}$ was trained on the Columbia track and had an issue in the middle of the training. The rest of the agents drove slowly and preferred to run out of time while almost standing still rather than crash. All the data provided was done with the hyperparameters that brought the best results.

7.7.2 Dreamer algorithm

The Dreamer algorithm performed more consistently, as can be seen from all of its trajectories. Mistakes consistently happened in the same places. The dreamer agent preferred to gather the reward from progressing the track and risking the crash. This behavior is preferable as the target was to complete a lap. Also, an expected behavior from the Dreamer agent is preferred to the inconsistent one of the TD3 agent. This way, if a change is made to the algorithm or the reward function, the changes in the driving can be assumed as the result.

Chapter 8
Future work

While providing some results and succeeding in successfully completing a lap with both *model-free* and *model-based* methods, there is a lot of work to drive faster and safer. We suggest ideas for improvements to the simulation environment and both model-free and model-based methods.

8.1 Environment and platform

While testing on the real car, we had to limit the speed because the model in the simulation was not accurate enough, as mentioned in Section 7.6. Before forcing the speed limit, the agent tried to steer and brake at higher speeds. There was not enough grip to change the velocity or heading.

Physics simulation. The simulator used in this thesis, the Racecar Gym [22], was built on the bullet physics engine. This engine is popular because it provides reasonably accurate physics simulations while the computing cost is still low. A more accurate simulator, like Project Chrono [44] or AWSIM [45], should help with minimizing the sim-to-real gap, which is discussed in Section 7.6.1. These simulators, however, are much more computationally expensive.

Accurate models. The car used in the simulation was the default car used in the Racecar Gym [22], with minor tweaks and changes. The accurate model was not used as the measurements, like the exact weight of individual parts or the wheel parameters, for our F1TENTH car have not yet been made. The Racecar Gym simulator is highly customizable, and modeling the car correctly and accurately should also help with a sim-to-real gap. The most important models and parameters for driving such a car are the wheel model and the surface model being driven on. If measured precisely, the car should handle traction levels in a real environment much better as the model used for the simulation will be more accurate.

8.2 Model-free agent

Software-related issues. The model-free agent has multiple flaws. Firstly, sometimes, it keeps returning the action of [1, 1], which means full throttle and full steer left. As it can be seen in Figs. 7.3 and 7.11, the agent gets stuck, refuses to progress in learning, and the reward drops to 0. Investigating the error was out of the scope of this work. We suspect the error can be in the implementation from Stable Baseline 3 [42] or its dependencies. It may come from the communication with the Racecar Gym [22] or in the simulator itself. However, the observations and rewards from the simulation have been checked and seem to be without a problem. In addition, the problem also happens in real-life experiments, namely with the Columbia and the Austria agents, where it did not communicate with the simulator but it communicated with the F1TENTH car through ROS2.

Upgrades from other methods. Apart from software-related issues, upgrades to the base implementation of the TD3 algorithm can be made. The inspiration can be taken from [15], where multiple approaches used in other areas of machine learning have been implemented to the DDPG algorithm. As TD3 is based on DDPG, these approaches should be applicable similarly. We mention and explain these improvements in the related work (Section 2.1.1). Also, the action smoothing from [14] should result in a smoother and faster drive.

8.3 Model-based agent

The Dreamer algorithm showed great potential. Nevertheless, the differences between the simulation and the real environment were too large. Some improvements could be used to achieve better performance. Many of them have already been implemented in the newer versions of the algorithm.

New versions. As stated in Section 2.1.1, in [19], the authors tried to use Dreamer v2 [18] to control a race car. However, the authors succeeded only in simulation and not on a real-life platform. In 2023, the third version of Dreamer was released [27]. The second or third version could be used to train the agent, as both methods show great potential in controlling the games used as baselines in the papers.

8.4 Utilizing additional information

In this thesis, we implemented two purely reactive end-to-end algorithms, TD3 and Dreamer, utilizing only the lidar as a sensor from the car. But as shown in Table 3.3, the platform also provides an IMU. However, the measurements from the IMU are rather noisy. If the IMU sensor inputs are filtered from the noise, they could provide valuable information that the RL models could utilize. The localization and a map can also be introduced, although it would not be full end-to-end, only partial, as described in Section 4.2. This should help with problems like in Figs. 7.9b and 7.10b, as the agent would know the map ahead without observing it with the lidar. The trajectory to help the learning could be added as in [13]. This, however, could introduce the same problems as mentioned in Section 2.1.1, e.g., the need to compute the optimal trajectory or the fact that the agent will not perform as well on other tracks.

Chapter 9
Conclusion

In this thesis, we successfully implemented, trained, and evaluated both model-free and model-based end-to-end approaches to control an F1TENTH race car, described in Section 3.3. The simulation environment used for training is described in Section 3.2. We compare the agents' learning processes and their success on four different tracks in simulation and one in real life (Chapter 7).

The model-free method used in this thesis is the TD3 algorithm. The implementation was provided by the Stable Baselines 3 library. This library was used because it provides implementations as baselines for testing and comparing. We added a reward function that promotes faster lap times and discourages hard braking and standing still. The TD3 method was inconsistent in learning and evaluation scenarios. We argue that this unexpected behavior results from a software problem as discussed in Section 8.2. Despite these problems, the agent trained on the simulated CIIRC track successfully finished a lap in the real-life version of this track. The lap time was 0.19 s slower than the model-based one while 1.41 s faster than the reactive Follow The Gap with the given setup.

The model-based method used in this work was the Dreamer algorithm. The implementation of the algorithm from [17] is used, as it has already run on the F1TENTH platform. We added the same reward function as in the model-free approach. This method behaves and trains itself much more consistently, as can be seen in Chapter 7. However, it drove a similar lap time to the TD3 agent in real-life scenarios. We argue that this was due to an inaccurate model and floor setup in the simulator as mentioned in Section 8.1. This led to limiting the maximal speed of the agent, as it could not overcome the sim-to-real gap. The lap time was 1.6 s faster than the Follow The Gap agent provided.

Both methods have shown potential as they managed to complete the laps. The times were even faster than the FTG algorithm, which was beyond our expectations. With further improvements discussed in Chapter 8, the lap times should get even faster and the agents should behave more consistently. We expect that the future of autonomous driving will utilize more Reinforcement Learning. Maybe not as full end-to-end agents, but in partial end-to-end approaches as this way, the RL agents could utilize modules for mapping, localization, or preprocessing the sensor input. Furthermore, the models trained by the model-based agents might be used in approaches like Model Predictive Control or other approaches that are highly dependent on accurate models. This way, the advantages of classical control would support the advantages of machine learning.

Bibliography

[1] F1TENTH. *F1TENTH - Build Documentation*. `https://f1tenth.org/build.html`. Accessed: 2024-30-04. 2023.

[2] Volkan Sezer and Metin Gokasan. "A novel obstacle avoidance algorithm:"Follow the Gap Method"". In: *Robotics and Autonomous Systems* 60.9 (2012), pp. 1123–1134.

[3] R. Craig Coulter. *Implementation of the Pure Pursuit Path Tracking Algorithm*. Tech. rep. CMU-RI-TR-92-01. Pittsburgh, PA, Jan. 1992.

[4] *University of Stuttgart, Model predictive control*. 2022. URL: `https://www.ist.uni-stuttgart.de/research/group-of-frank-allgoewer/model-predictive-control/`.

[5] Scott Fujimoto, Herke Hoof, and David Meger. *Addressing function approximation error in actor-critic methods*. PMLR, 2018.

[6] Danijar Hafner et al. *Dream to control: Learning behaviors by latent imagination*. 2019.

[7] Dominik Hodan. *Reinforcement learning-based control system for the SK8O robot*. 2023.

[8] Johannes Betz et al. "Autonomous Vehicles on the Edge: A Survey on Autonomous Vehicle Racing". In: *IEEE Open Journal of Intelligent Transportation Systems* 3 (2022), pp. 458–488. DOI: `10.1109/OJITS.2022.3181510`.

[9] Alex Kendall et al. "Learning to Drive in a Day". In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 8248–8254. DOI: `10.1109/ICRA.2019.8793742`.

[10] *DDPG algorithm description*. 2014. URL: `https://spinningup.openai.com/en/latest/algorithms/ddpg.html#background`.

[11] Florian Fuchs et al. "Super-Human Performance in Gran Turismo Sport Using Deep Reinforcement Learning". In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 4257–4264. DOI: `10.1109/LRA.2021.3064284`.

[12] B Evans et al. "Accelerating online reinforcement learning via supervisory safety systems". In: *arXiv preprint arXiv:2209.11082* (2022).

[13] Benjamin David Evans, Herman Arnold Engelbrecht, and Hendrik Willem Jordaan. *High-speed autonomous racing using trajectory-aided deep reinforcement learning.* 2023.

[14] Zhijie Lu et al. "Deep Reinforcement Learning Based Autonomous Racing Car Control With Priori Knowledge". In: *2021 China Automation Congress (CAC).* 2021, pp. 2241–2246. DOI: `10.1109/CAC53003.2021.9728289`.

[15] Adrian Remonda et al. *Formula rl: Deep reinforcement learning for autonomous racing using telemetry data.* 2021.

[16] Peide Cai et al. *Vision-based autonomous car racing using deep imitative reinforcement learning.* 2021.

[17] Axel Brunnbauer et al. *Latent imagination facilitates zero-shot transfer in autonomous racing.* IEEE, 2022.

[18] Danijar Hafner et al. *Mastering atari with discrete world models.* 2020.

[19] Tanay Dwivedi et al. "Continuous Control of Autonomous Vehicles using Plan-assisted Deep Reinforcement Learning". In: *2022 22nd International Conference on Control, Automation and Systems (ICCAS).* 2022, pp. 244–250. DOI: `10.23919/ICCAS55662.2022.10003698`.

[20] Radoslav Ivanov et al. "Case Study: Verifying the Safety of an Autonomous Racing Car with a Neural Network Controller". In: *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control.* HSCC '20. Sydney, New South Wales, Australia: Association for Computing Machinery, 2020. ISBN: 9781450370189. DOI: `10.1145/3365365.3382216`. URL: `https://doi.org/10.1145/3365365.3382216`.

[21] Kyle Stachowicz et al. "Fastrlap: A system for learning high-speed driving via deep rl and autonomous practicing". In: (2023), pp. 3100–3111.

[22] Axel Brunnbauer and Luigi Berducci. *racecar_gym.* Version 0.0.1. URL: `https://github.com/axelbr/racecar_gym`.

[23] Matthew O'Kelly et al. "F1TENTH: An Open-source Evaluation Environment for Continuous Control and Reinforcement Learning". In: *NeurIPS 2019 Competition and Demonstration Track.* PMLR. 2020, pp. 77–89.

[24] Erwin Coumans and Yunfei Bai. *PyBullet, a Python module for physics simulation for games, robotics and machine learning.* `http://pybullet.org`. 2016–2021.

[25] Mark Towers et al. *Gymnasium.* Mar. 2023. DOI: `10.5281/zenodo.8127026`. URL: `https://zenodo.org/record/8127025` (visited on 07/08/2023).

[26] Ardi Tampuu et al. “A Survey of End-to-End Driving: Architectures and Training Methods”. In: *IEEE Transactions on Neural Networks and Learning Systems* 33.4 (Apr. 2022), pp. 1364–1384. DOI: 10.1109/tnnls.2020.3043505. URL: https://doi.org/10.1109%2Ftnnls.2020.3043505.

[27] Danijar Hafner et al. “Mastering diverse domains through world models”. In: *arXiv preprint arXiv:2301.04104* (2023).

[28] Oriol Vinyals et al. “Grandmaster level in StarCraft II using multi-agent reinforcement learning”. In: *Nature* 575.7782 (Nov. 2019), pp. 350–354. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1724-z. URL: https://doi.org/10.1038/s41586-019-1724-z.

[29] Christopher Berner et al. *Dota 2 with large scale deep reinforcement learning*. 2019.

[30] Brenna D. Argall et al. “A survey of robot learning from demonstration”. In: *Robotics and Autonomous Systems* 57.5 (2009), pp. 469–483. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2008.10.024. URL: https://www.sciencedirect.com/science/article/pii/S0921889008001772.

[31] Felipe Codevilla et al. *Exploring the limitations of behavior cloning for autonomous driving*. 2019.

[32] Connor Shorten and Taghi M. Khoshgoftaar. “A survey on Image Data Augmentation for Deep Learning”. In: *Journal of Big Data* 6.1 (July 2019), p. 60. ISSN: 2196-1115. DOI: 10.1186/s40537-019-0197-0. URL: https://doi.org/10.1186/s40537-019-0197-0.

[33] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. *A reduction of imitation learning and structured prediction to no-regret online learning*. JMLR Workshop and Conference Proceedings, 2011.

[34] et al. Richard S Sutton Andrew G Barto. *Reinforcement learning, An Introduction, second edition*. The MIT Press, 2018.

[35] Yunlong Song et al. “Autonomous Overtaking in Gran Turismo Sport Using Curriculum Reinforcement Learning”. In: *CoRR* abs/2103.14666 (2021). arXiv: 2103.14666. URL: https://arxiv.org/abs/2103.14666.

[36] Wah Loon Keng Laura Graesser. *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*. Addison-Wesley Data & Analytics Series. Addison-Wesley, 2020. ISBN: 0135172381; 9780135172384. URL: libgen.li/file.php?md5=202217fd5fd079d3910c39a6db4d0598.

[37] Christopher JCH Watkins and Peter Dayan. “Q-learning”. In: *Machine learning* 8 (1992), pp. 279–292.

[38] Ngan Le et al. “Deep Reinforcement Learning in Computer Vision: A Comprehensive Survey”. In: *CoRR* abs/2108.11510 (2021). arXiv: 2108.11510. URL: https://arxiv.org/abs/2108.11510.

[39] Nicolas Grossmann, Eduard Gröller, and Manuela Waldner. "Concept splatters: Exploration of latent spaces based on human interpretable concepts". In: *Computers & Graphics* 105 (2022), pp. 73–84. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2022.04.013. URL: https://www.sciencedirect.com/science/article/pii/S0097849322000656.

[40] *TD3 algorithm description.* 2018. URL: https://spinningup.openai.com/en/latest/algorithms/td3.html.

[41] Timothy P Lillicrap et al. *Continuous control with deep reinforcement learning.* 2015.

[42] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[43] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32.* Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[44] Project Chrono. *Chrono: An Open Source Framework for the Physics-Based Simulation of Dynamic Systems.* http://projectchrono.org.

[45] Tier IV. *AWSIM.* https://tier4.github.io/AWSIM/.