



BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS
FACULTY OF MECHANICAL ENGINEERING
DEPARTMENT OF MECHATRONICS, OPTICS, AND MECHANICAL
ENGINEERING INFORMATICS

Deep reinforcement learning methods for the motion control design of the F1TENTH electric vehicle: bridging the sim-to-real gap

Csanád Budai and Tamás Széles

Scientific Conference of Students

Supervisors:

Dr. Csaba Budai
Associate Professor
BME GPK MOGI

Dr. Balázs Németh
Senior research fellow
SZTAKI Systems and Control Lab

BUDAPEST, 2024

Abstract

Intelligent robotics systems have fascinated engineers and researchers for a long time, and the past few years have been outstandingly fruitful in their development, solving problems that have previously seemed impossible. A strong force pushing this growth has been the advancement of deep reinforcement learning, an approach that enables autonomous agents to discover their environments and use this knowledge to complete a given goal highly effectively.

However, despite all the progress, there are still many challenges limiting the widespread application of these techniques. Training in the real world is difficult because even the most effective schemes require huge amounts of data, which is both expensive and time-consuming to collect. To alleviate this, practitioners use simulators to train their agents, modeling the actual conditions. Unfortunately using a synthetic training environment leads to models exploiting the imperfections of the modeled dynamics, that do not work during live application. This is called the sim-to-real gap.

F1TENTH is an international competition of small-scale autonomous vehicles, with a focus on racing algorithms, as hardware is identical. Our research was originally motivated by this challenge, which also provided a great opportunity to use learning algorithms, as the race required strong performance and complex decision-making.

In this work, we discuss our process of creating an agent that effectively competed in the race, from developing our simulator from scratch to bridging the sim-to-real gap. To facilitate the real-world transfer, we did not create a perfect synthetic environment, but rather used methods like domain randomization, learning with disturbances, and clever state-aggregation. We carried out an ablation study on an F1TENTH vehicle showing that every element of our sim-to-real transfer method is important. To gain a better understanding of the behavior of our final model, we conducted additional tests. Furthermore, we provide an introduction into the theoretical background of deep reinforcement learning and review the current state-of-the-art literature on real-world robotics learning systems. Finally, we discuss some of the future directions for creating more adaptive and data-efficient algorithms that enable smarter decision-making.

Kivonat

Az intelligens robotok már régóta lenyűgözik a mérnököket és kutatókat. Kiemelkedően eredményes volt az elmúlt néhány év ezen rendszerek fejlesztésében, több olyan feladat lett megoldva, melyek korábban lehetetlennek tűntek. A fejlődés egyik nagy mozgatórugója a mély megerősítéses tanulás módszere, ami lehetővé teszi az autonóm ágenseknek, hogy a környezetük felfedezésével önállóan fejlődjenek.

Azonban, a nagy előrelépések ellenére, még mindig sok kihívás hátráltatja ezen módszerek széleskörű elterjedését. Nincsen egyszerű dolgunk, hogyha a valóságban szeretnénk tanítani, mert még a leghatékonyabb technikák is hatalmas mennyiségű adatot igényelnek, aminek nagy a költsége és sokszor tekintélyes időbe kerül. Az erőforrásigények csökkentéséhez gyakran használatosak a valóságot leképező szimulátorok. Sajnos előfordul a szintetikus környezetben tanításkor, hogy az ágensek a modellezett dinamika pontatlanságait kihasználják és így a szimulátorban erősnek tűnő modellek a valóságban nem lesznek hatékonyak. Ezt nevezzük a sim-to-real gap-nek.

Az F1TENTH egy nemzetközi kisméretű önvezető autóverseny, ahol az azonos hardver miatt csak az algoritmusok közötti különbségek döntenek. Kezdetben ez a megmérettetés motiválta kutatásunkat, emellett a tanulási módszerek megismerésére is szintén nagyon jó lehetőséget nyújtott, hiszen az eredményes versenyzéshez magasfokú teljesítmény és komplex döntéshozatal szükséges.

A dolgozatunkban bemutatjuk, hogy miként készítettünk egy olyan megerősítéses tanulási ágenszt, ami képes volt valós környezetben versenyezni. Ehhez ismertettük az általunk írt szimulátor elkészítésének, valamint a sim-to-real gap áthidalásának lépéseit. Az áthidalást nem a szimuláció tökéletesítésével, hanem domain randomization, learning with disturbances, valamint sajátos state-aggregation segítségével oldottuk meg. Azért, hogy bemutassuk a technikánk elemeinek fontosságát, végeztünk egy eliminációs vizsgálatot is. További teszteket hajtottunk végre a végső modellünk megismerésének céljából. Ezeken felül betekintést nyújtunk a megerősítéses tanulás elméleti alapjaiba, illetve áttekintjük a legújabb tanuló rendszerekkel foglalkozó szakirodalmat. Végül kitérünk a kutatásunk lehetséges jövőbeli irányaira, az adaptív, adathatékony algoritmusokra, amikkel intelligensebb döntéshozó rendszereket lehetne fejleszteni.

Contents

1	Introduction	3
1.1	Problem setting and motivation	3
1.2	State-of-the-art	4
1.3	Proposed solution	5
2	Preliminaries	6
2.1	Reinforcement Learning	6
2.1.1	Basic concepts in reinforcement learning	6
2.1.2	Q-Learning	9
2.1.3	Exploration-Exploitation dilemma	10
2.1.4	Curse of dimensionality	11
2.1.5	Sparse reward problem	11
2.1.6	Credit assignment problem	11
2.2	Deep Learning	11
2.2.1	The perceptron	12
2.2.2	The Multilayer Perceptron	13
2.2.3	What characterizes a good mapping?	14
2.2.4	Stochastic Gradient Descent	15
2.2.5	Imitation Learning	18
2.3	Deep Reinforcement Learning	19
2.3.1	Deep Q-Networks	20
2.3.2	Policy Gradient	21
2.4	The sim2real problem	22
2.5	Open-source platforms	23
2.5.1	<code>gymnasium</code> module	23
2.5.2	<code>stable-baselines3</code> module	24
2.5.3	F1TENTH platform	24
2.6	Dynamics model	25
3	Our contribution	26

3.1	Development of the simulator	26
3.1.1	Tracks	27
3.1.2	Getting the starting state	28
3.1.3	Transition function	28
3.1.4	Control signal	29
3.1.5	Checkpoint and collision detection	29
3.1.6	Generating the LiDAR reading	29
3.2	Development of the RL training stack	29
3.2.1	Our problem in terms of RL	29
3.2.2	Model structure	31
3.2.3	Reward mechanisms and training scheme	33
3.3	Bridging the sim-to-real gap	34
4	Experimental results	35
4.1	Tested policies	36
4.2	Test tracks	37
4.3	Simulator testing	37
4.4	Real-world testing	39
5	Racing against humans	39
6	Conclusion	40
7	Future Work	41
7.1	Adaptive RL	41
7.2	Sample-efficient RL	42
8	Acknowledgments	43
8.1	Hyperparameters for PPO	48
8.2	The constants used for Pacejka’s Magic formula	48

1 Introduction

1.1 Problem setting and motivation

The F1TENTH international competition [1] is a race of mechanically identical autonomous small-scale cars, making it a race purely of algorithms. To compete, a team must create an algorithm that can safely and quickly race around the track, while being able to stop at any moment if an obstacle appears. The track is bounded by a continuous wall of ventilation tubes, forming a loop with many difficult turns and slippery surfaces. After the demonstration of safe driving, and the ability to avoid static and dynamic obstacles, the race begins. Teams first compete in safety and speed in the qualification round, which is followed by an intense head-to-head race of two cars, paired based on the qualification.

Thus for our goal of effective competition in the race, we needed our algorithm to be able to do the following:

- Complete the lap as **quickly and safely** as possible.
- **Make informed decisions:** The car must choose its control inputs taking future events into account, **preparing** for the movement of other vehicles and to **overtake** in head-to-head.
- Do all of this **without localization or mapping:** We observed from past competitions, that cars often fail to localize after the track has been slightly rearranged by crashes.
- Perform well under **diverse conditions:** track size, friction obstacles often change within a single lap.

We found this challenge exciting because the tasks that need to be solved are much more general than just the competition. In general, the key capabilities of any intelligent robotics system are similar: they must also complete their task safely and effectively, and make smart decisions in previously unseen situations while being resilient to environmental changes. We also wanted to deepen our knowledge of learning systems and apply the theory in practice. For this, the F1TENTH competition proved to be the perfect opportunity.



Figure 1.1: Human vs. AI F1TENTH event at the Kármán Dormitory

1.2 State-of-the-art

Many teams prepare to compete at the international F1TENTH events. The most commonly used techniques are global planning and reactive methods.

To perform global planning, one needs to first create a precise mapping of the track using SLAM [2]. Next, a racing line must be calculated, which when followed results in optimal lap time. Different methods, such as geometric and genetic algorithms are deployed. Finally, the car needs to progress along this path, as precisely as possible. To do this, the car must know its current position in the global frame, and a low-level control algorithm needs to be used to stay on the racing line, such as Model Predictive Control with Constraints [3], and Pure Pursuit [4]. This requires precise and robust tracking and localization. Overtaking is an additional challenge, which requires the development of extra methods for recognizing the other car, deciding when to overtake, and then performing the maneuver correctly. One of the best teams from Zurich made their racing stack open-source: [5] provides a great overview of their implementation of the global planning method.

Reactive methods have ruled the competition for a long time, most winning teams employ the Follow The Gap (FTG) method, also often called the Disparity Extender algorithm [6]. This method targets the furthest point seen by the car every timestep, basically always moving in the direction of the largest hole, which the car can fit through. Almost all teams use this, as it is reliable and simple to implement. The most commonly used controller for this is Pure Pursuit.

In the past, few teams have attempted to use learning-based methods, mainly because of their unreliability. The teams that did end up trying were only able to safely race at speeds well under 2m/s, which is very slow compared to others, the best teams reaching 6m/s and beyond.

The use of learning-based methods, particularly Reinforcement Learning (RL), is a common topic of academic interest within the field of robotics and control. The strength of these approaches has been demonstrated time and again: researchers were able to teach a robot dog to walk from nothing in just a couple of minutes [7], and [8] was able to learn how to solve a Rubik’s Cube with one hand. Another exciting feat was the creation of football robots that learned to play from experience in a simulator [9] and then transferred their skills into a real environment.

In spite of these benefits, the efficient performance of reinforcement learning has challenges. Even when paired with strong compute, RL algorithms can still struggle to learn because of the curse of dimensionality (see Section 2.1.4). Because of this RL requires lots of data that can only be collected over a high number of episodes. In the case of automotive control systems, the training process is performed using simulators that approximate the real vehicle’s dynamics.

Thus, the process of applying RL consists of first training the model in a simulator, and then deploying it in the real world. Unfortunately, using a synthetic training environment can lead to overfitting onto the modeled dynamics (see Section 2.2.3), i.e., the trained controller learns to exploit elements of the simulator that are non-existent in the real world. RL-based controllers developed in this way, often can not operate with high efficiency under real-world scenarios. This is called the sim-to-real gap (see [10] and Section 2.4), which can be handled with different methods, such as real-world fine-tuning, training with disturbances, and domain randomization [11].



(a) Roborace self-driving competition



(b) Head-to-head F1TENTH racing

Figure 1.2: Autonomous racing competitions

1.3 Proposed solution

In this work, we propose a fully reactive, reinforcement learning based controller, that can execute complex decisions under various, challenging conditions. To meet these criteria

we chose to use RL, because of its ability to make decisions and come up with unexpected solutions to unseen situations. Another reason to do RL was that we found this technology quite interesting and we were motivated by its advancements in other areas of engineering. We show the full process of creating the autonomous racing agent.

Our first main contribution is the development of our own simulator, which plays a crucial role during training. We thoroughly describe all parts of the simulator, which makes reproducing our work possible. Our simulator includes support for various customizations, making possible the elements we need for bridging the gap and improving the capabilities of our trained models. The simulator supports custom model structure, novel preprocessing, and various dynamic training environments.

The central point of this work is our approach to bridging the sim-to-real gap. We describe the additional measures we took to enable a seamless transition, the main components being:

- **Domain randomization:** The perturbation of the training environment to prepare the agent for diverse situations.
- **State aggregation:** The use of an observation preprocessing technique to enable fast learning and disrupt simulator overfit.
- **Learning with disturbances:** The spawning of dynamic obstacles that force the agent to practice the avoidance and overtaking of other cars.

Agents trained using the full RL stack were tested out in different real-world and simulated scenarios measuring their robustness. Furthermore, we organized an event where humans controlling an F1TENTH vehicle using FPV goggles raced against our best models. With an early version of our stack, we have also participated in an F1TENTH competition in Yokohama, Japan, finishing 12th overall, being the only team with a working, learning-based algorithm. From the findings of our work, we submitted a conference paper to the American Control Conference 2025. During this work we did not use ANY text or code written by ChatGPT, all of the illustrations were created by us.

We created a short video demonstrating the different elements of our method during real-world application: <https://youtu.be/1P7mlawgVDo>

2 Preliminaries

2.1 Reinforcement Learning

2.1.1 Basic concepts in reinforcement learning

The creation of truly self-learning systems is a central goal for most AI researchers. Reinforcement learning provides a biology-inspired approach to this. The main idea of

reinforcement learning comes from the fact, that a truly self-learning agent should be able to learn from just the interactions with its environment. Even toddlers learn from trying out certain things in their surroundings, testing environment responses. Reinforcement learning techniques try to capture this idea, by creating an agent that receives some observations from the environment (camera feed, LiDAR data, previous stock prices) every timestep and returns an action (actuator positions, buy/sell signal), after which the environment gives the agent the next observation and a reward, showing the agent how well it executed the task. It is important to note that, these rewards are in general not directly correlated with the quality of the last decision. The agent has the goal of receiving as much reward as possible, i.e. maximizing the cumulative reward over some horizon. With this seemingly simple idea researchers were able to solve various complex robotics and decision-making problems, such as learning to control a robotic dog [12] or solving a Rubik’s cube with one hand [8].

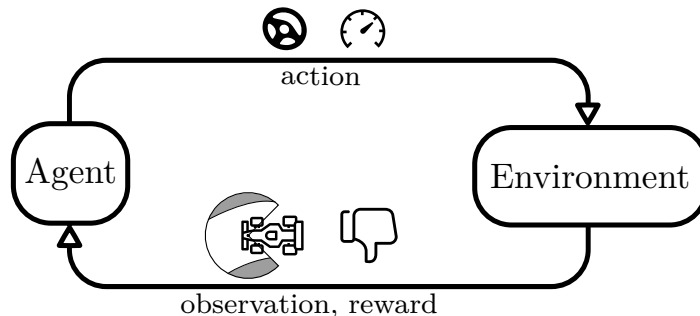


Figure 2.1: Agent-environment interaction

Most reinforcement learning algorithms formalize the RL problem as a Markov Decision Process (MDP), which is defined by the $(\mathcal{S}, \mathcal{A}, p, \mathcal{R})$. In MDP-s, it is assumed that the agent receives the full state as an observation. MDP-s where the agent receives just a part of the state are called Partially Observable MDP-s (POMDP). The theory of POMDP-s will not be discussed here, because almost all modern algorithms assume that the environment is an MDP (they work quite well in the POMDP setting too). The RL notations and definitions used in this work are from [13]. Note that upper-case variable names mean random variables. $\mathcal{S}$ is the state space, it characterizes the possible states of the environment (possible board states, possible stock values). In MDP-s the Markov property holds, meaning that the next state only depends on the current state and the action taken in the current state (the system is memoryless). $\mathcal{A}$ is the action space, the space of all possible actions, that the agent can take (possible turn angles, buy/sell signal). $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ is the dynamics function, which shows the probability of transitioning from state $s \in \mathcal{S}$ with action $a \in \mathcal{A}$ to state $s' \in \mathcal{S}$ and receiving reward $r \in \mathcal{R}$. $\mathcal{R} \subset \mathbb{R}$ is the space of possible immediate rewards. It is good practice to have $\mathcal{R} = [0, 1]$ because this way it is easier to compare two different policies, and library implementations tend to assume that the reward is in this range.

$$p(s', r | s, a) \equiv \text{P}\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\} \quad (2.1)$$

The classical view on RL is to assume a finite horizon setting, meaning that the number of steps taken in a single episode is bounded (at most T). So formally the goal of an RL algorithm is that given any state $s \in \mathcal{S}$ find the optimal action $a \in \mathcal{A}$ that maximizes the expected cumulative reward G (Equation 2.2). In other words, the RL algorithm needs to produce a policy that gathers the most reward possible from any given state. However, this naive summarization of the rewards makes learning infinite-horizon problems impossible. To resolve this issue discounting is used, that is the value of the future rewards is smaller than immediate rewards. Equation 2.3 shows the objective which the agent seeks to maximize, in the infinite-horizon setting, where $\gamma \in [0, 1]$ is the discount factor.

$$G_t \equiv \mathbb{E} \left[\sum_{k=0}^T R_{t+k+1} \right] \quad (2.2)$$

$$G_t \equiv \mathbb{E} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \right] \quad (2.3)$$

A policy could be any function $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$. Here $\pi(a|s)$ shows the probability of taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$. Classical RL algorithms had π as a lookup table and from now on π is a lookup table here as well, but calculating such a table can be a challenging task even on small problems. To solve this problem, the idea of a value function $v_\pi : \mathcal{S} \rightarrow \mathbb{R}$ is introduced. The value function shows the expected return from a certain state s if the agent follows policy π thereafter (Equation 2.4). $\mathbb{E}_\pi[\cdot]$ means the expected value of a random variable if the agent follows policy π and t is any time step. The value function is also called the state-value function for policy π . It is also useful to define an action-value function, also named q function, that specifies the expected return from a state s , taking action a and following policy π thereafter. These two functions are central in reinforcement learning and most classical, dynamic programming based RL algorithms try to learn these functions.

$$\begin{aligned} v_\pi(s) &\equiv \mathbb{E}_\pi [G_t \mid S_t = s] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \end{aligned} \quad (2.4)$$

$$\begin{aligned} q_\pi(s, a) &\equiv \mathbb{E}_\pi [G_t \mid S_t = s, A_t = a] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \end{aligned} \quad (2.5)$$

The optimal state-value function (v^*) and action-value function (q^*) are the v and q functions for the optimal policy π^* . The optimal value functions need to satisfy the Bellman optimality equations (Equation 2.6 and Equation 2.7). Given v^* and a perfect environment model, the optimal policy greedily chooses the action that has the highest expected value in the next state. Similarly, given q^* the optimal policy takes the action

with the highest q value in the current state. Because a q^* based policy does not need a perfect environment model, most value-based algorithms (methods that implicitly store the policy as a state-value or action-value function) use the q function.

$$v^*(s) = \max_a \mathbb{E} [R_{t+1} + \gamma v^*(S_{t+1}) \mid S_t = s, A_t = a] \quad (2.6)$$

$$q^*(s, a) = \mathbb{E} \left[R_{t+1} + \gamma \max_{a'} q^*(S_{t+1}, a') \mid S_t = s, A_t = a \right] \quad (2.7)$$

There is a categorization of RL algorithms that has not yet been touched upon. Off-policy and on-policy RL algorithms differ in the type of training data they can learn on. Off-policy algorithms can use data gathered by other policies (possibly the past version of a policy) as long as there is sufficient overlap between the policies. On-policy algorithms on the other hand can only use data that they generated themselves. This shortcoming limits the use of past experiences, which leaves a mark on sample efficiency.

2.1.2 Q-Learning

Learning either one of the optimal value functions is no easy task and many algorithms were proposed to solve this problem, but here only Q-Learning [14] will be shown, because of its historical importance. The goal of the Q-Learning algorithm is to iteratively learn the optimal action-value function q^* . The idea here is to use one of the Bellman optimality equations (Equation 2.7) to construct the update rule (Equation 2.8) for the policy. The update rule updates the current policy (a lookup table for the q function) at every timestep. In the tabular setting (where the q function is represented as a lookup table for every state-action pair) Q-learning is proven to converge to the optimal solution. α is the learning rate for the policy. The term $R + \gamma \max_{a'} Q_t(s', a') - Q_t(s, a)$ is called the 1-step temporal-difference (TD) error because it shows the error of the policy after 1 timestep. There are algorithms that calculate N step TD error, which enables faster propagation of the reward, which improves data efficiency and alleviates the sparse-reward problem (Section 2.1.5). To solve the exploration-exploitation dilemma (Section 2.1.3) ε – greedy exploration is used. This simply means taking a random action with ε probability at every timestep and following the action-value function otherwise. On problems with small action spaces (not many actions to choose from), this exploration strategy can work quite well, but as the size of the problem increases, the need for a more sophisticated exploration technique also rises. The Q-learning algorithm is shown in Algorithm 1.

$$Q_{t+1}(S_t, A_t) \leftarrow Q_t(S_t, A_t) + \alpha \left[R_t + \gamma \max_{a'} Q_t(S_{t+1}, a') - Q_t(S_t, A_t) \right] \quad (2.8)$$

Algorithm 1 Q-Learning Algorithm

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}$ arbitrarily except that $Q(\text{terminal}, \cdot) = 0$
for each episode **do**
 Initialize state S
 while not terminal **do**
 Choose action A from state S using policy derived from Q (ϵ -greedy)
 Take action A , observe reward R , and next state S'
 Update the Q-value:
 $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \max_{a'} Q(S', a') - Q(S, A) \right]$
 Update state $S \leftarrow S'$
 end while
end for

2.1.3 Exploration-Exploitation dilemma

Given an agent that already knows some things about its environment, how should it choose its actions? Take suboptimal actions (according to the agent's current beliefs) or take the optimal action (based on the current knowledge of the agent)? This problem is called the exploration-exploitation dilemma and is one of the hardest problems in reinforcement learning. Efficient exploration strategies enable agents faster learning and better convergence. Some techniques that are trying to solve this problem:

- **ϵ -greedy exploration:** Choose a random action with ϵ probability at every timestep, take the supposed-optimal action with $1 - \epsilon$ probability
- **Softmax exploration:** Sample randomly from all the actions based on the associated values (higher-valued actions have higher probabilities).
- **Intrinsic motivation based exploration** [15], [16]: The agent receives rewards for reaching interesting states.
- **Count-Based exploration** [17]: The agent gets rewards for visiting rarely visited states.
- **Optimistic Initialization** [18]: Instead of initializing the Q-table with random values, initialize it with high initial value estimates everywhere. This way the agent "thinks" that unseen states have high returns and explores them.
- **Upper Confidence Bound based exploration** [19], [20]: The value for each state is augmented with a confidence bonus, that becomes smaller as the agent visits the state more.

2.1.4 Curse of dimensionality

Early reinforcement learning algorithms did not use any type of function approximation to store the policy, so as $|\mathcal{S} \times \mathcal{A}|$ (the size of the combined state-action space) increases, the size of the policy also increases. These tabular policies were only usable in small problems, limiting their practical applicability. Modern algorithms, with the use of function approximators, can handle bigger state spaces, but learning in high-dimensional spaces still poses a hard problem, mostly because it makes the exploration-exploitation dilemma (Section 2.1.3) much harder. Modern algorithms alleviate this issue by learning compact representations that transform the problem into a lower dimensional space, where it is easier to learn.

2.1.5 Sparse reward problem

RL algorithms adjust their policy based only on rewards, but when this reward is sparse the learning is significantly slowed down, if not completely impossible. Exploration, the backbone of learning, becomes infeasible without any intrinsic motivation, so vanilla RL agents fail to solve problems with sparse rewards. The solution here is to artificially make the reward much more dense, providing a better learning signal, or to create an RL algorithm that has its own internal reward function and can explore interesting states based on that.

2.1.6 Credit assignment problem

The main problem of RL is to learn which actions are good decisions (in the long run) in certain states. In other words from all the actions, the agent needs to first learn the value of each action to then take the supposed optimal one. Based on just sampled trajectories, it is quite hard to judge the value of a certain action in a certain state. This is called the credit assignment problem. Sparse rewards make this problem much worse. In general, value-based RL algorithms often try to solve this problem.

2.2 Deep Learning

Deep Learning (DL) is the method of using an Artificial Neural Network (ANN) as a universal function approximator. An ANN is a mathematical object made up of neuron-like objects called perceptrons. These perceptrons are connected to form a network of neurons with some of them acting as inputs and outputs. One of the widespread methods of DL is the Multilayer Perceptron (MLP), where the network is made up of fully connected layers of artificial neurons. The MLP is a powerful tool for supervised learning, where the task is to learn a good mapping function f on the dataset $\mathcal{D}$. In this work, deep learning notation is used as in [21].

2.2.1 The perceptron

The perceptron (first described by [22]) is an artificial neuron that takes some k inputs, and gives a single output according to some rule, the structure of which is described below. The output can later be sent as an input to any amount of other neurons. Figure 2.2 shows the mechanism inside of each perceptron.

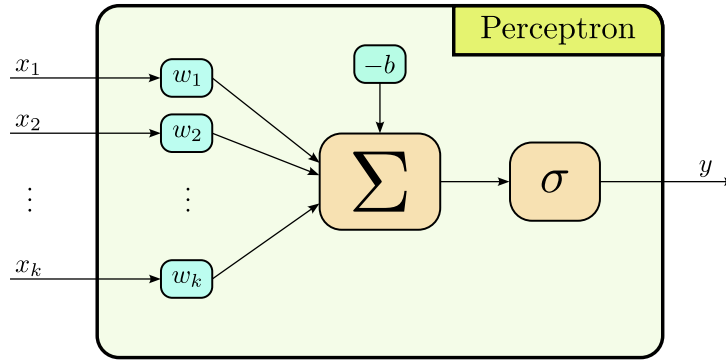


Figure 2.2: Perceptron structure

1. The k inputs of the perceptron $x_1, \dots, x_k$ are individually weighted by weights $w_1, \dots, w_k$, and added together. This is because of the idea, that a neuron can be more sensitive to some inputs (higher $|w|$), and less sensitive to others (low $|w|$). For example, our brain concentrates more on what is in front of our eyes, compared to our side.
2. A bias b is added to the sum. This is inspired by biology, there is a minimum input needed to push the output to be positive. Neurons spike if they are triggered enough, but there is always a minimum bound.
3. A non-linear, so-called activation function σ is applied. This is based on the idea, that double the intensity of a stimulation does not cause double the response. Real neurons also do not double their output for double the input. A commonly used non linearity is ReLU $\sigma_{\text{ReLU}}(x) = \max(0, x)$.

The relationship between the input $x \in \mathbb{R}^k$ and output $y \in \mathbb{R}$ of a function is:

$$y = \sigma \left(\sum_{i=1}^k x_i \cdot w_i + b \right) \quad (2.9)$$

Let $\mathbf{w}$ be the row vector of weights, with the first value being b . Now if we add 1 to the beginning of the input vector and use $\mathbf{x}^T = [1, x_1, x_2, \dots, x_k]$, we can simplify the equation:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x}) \quad (2.10)$$

This output y is often called the activation of the perceptron. The parameters in a perceptron that can be changed are θ , being $w_1, \dots, w_k$, and b .

2.2.2 The Multilayer Perceptron

The MLP is made up of layers of perceptrons fully connected to their neighboring layers. A denotes the number of inputs, and B the number of outputs, l_i the number of neurons in layer i , $\mathbf{x}^{(i)}$ the input of layer i , $\mathbf{y}^{(i)}$ the output of layer i , $\mathbf{w}_j^{(i)}$ the weight vector of perceptron j on layer i . Figure 2.3 shows the feed-forward structure of a general MLP.

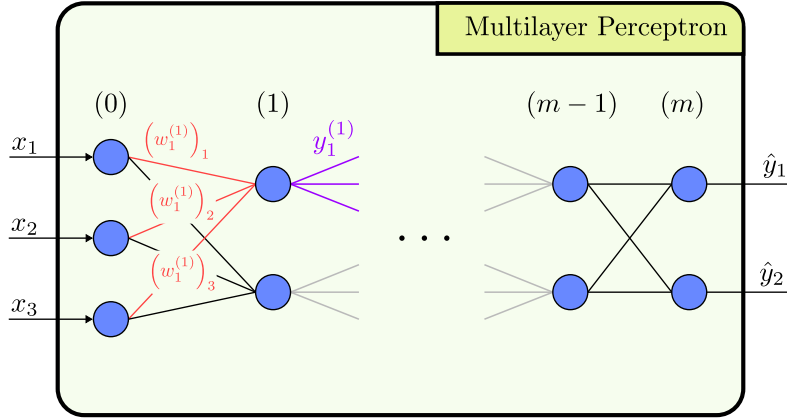


Figure 2.3: Multilayer Perceptron structure

The input layer (shown as layer (0) in Figure 2.3) corresponds with the input.

The first hidden layer is made up of l_1 neurons, where each neuron has the input $\mathbf{x}^{(1)} = \mathbf{x} \in \mathbb{R}^A$ from layer (0). Each neuron produces a single numerical output, thus the output of neuron j can be written as:

$$y_j^{(1)} = \sigma(\mathbf{w}_j^{(1)} \cdot \mathbf{x}^{(1)}) \quad (2.11)$$

Or the output can be written as a vector:

$$\mathbf{y}^{(1)} = \sigma(\mathbf{W}^{(1)} \cdot \mathbf{x}^{(1)}) \quad (2.12)$$

Where $\mathbf{y}^{(1)} \in \mathbb{R}^{l_1}$ is the output vector of layer (1), $\mathbf{W}^{(1)}$ denotes the matrix of weights, each row referring to the weights of one perceptron $\mathbf{W}_{j*}^{(1)} = \mathbf{w}_j^{(1)}$, and σ is used piecewise.

The outputs of a layer are always fed to be inputs of the next one.

$$\mathbf{y}^{(i)} = \sigma(\mathbf{W}^{(i)} \cdot \mathbf{x}^{(i)}) \quad (2.13)$$

$$\mathbf{y}^{(i)} = \mathbf{x}^{(i+1)} \quad (2.14)$$

$$\mathbf{y}^{(i+1)} = \sigma(\mathbf{W}^{(i+1)} \cdot \mathbf{y}^{(i)}) \quad (2.15)$$

Therefore if there are m layers, the feed-forward can be written as:

$$\mathbf{y}^{(1)} = \sigma(\mathbf{W}^{(1)} \cdot \mathbf{x}) \quad (2.16)$$

$$\mathbf{y}^{(2)} = \sigma(\mathbf{W}^{(2)} \cdot \mathbf{y}^{(1)}) \quad (2.17)$$

...

$$\mathbf{y}^{(m-1)} = \sigma(\mathbf{W}^{(m-1)} \cdot \mathbf{y}^{(m-2)}) \quad (2.18)$$

$$\mathbf{y} = \mathbf{y}^{(m)} = \sigma(\mathbf{W}^{(m)} \cdot \mathbf{y}^{(m-1)}) \quad (2.19)$$

The final layer (m) is called the output layer, its activations represent the output of the whole network. Therefore if we separate f into functions representing each layer $f^{(i)}$:

$$\mathbf{y} = f_{\theta}^{(m)}(f_{\theta}^{(m-1)}(\dots f_{\theta}^{(1)}(\mathbf{x}))) \quad (2.20)$$

Or, all of the steps can be simplified as:

$$\mathbf{y} = f_{\theta}(\mathbf{x}) \quad (2.21)$$

Where θ denotes all of the parameters of the network $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \dots$

2.2.3 What characterizes a good mapping?

As mentioned previously f is created with the goal of being a good mapping on the labeled dataset $\mathcal{D}$ of pairs of inputs $\mathbf{x}$ and outputs $\mathbf{y}$.

$$\mathcal{D} = (\mathbf{x}, \mathbf{y})_{i=0}^N \quad (2.22)$$

We want to describe what our optimal function would look like. We want f to work well for $\mathcal{D}$, but our real goal is to be able to estimate $\mathbf{y}$ for values of $\mathbf{x}$ that we have not seen before. This is called generalization. A first guess could be to find θ where $f_{\theta}(x_i) = y_i \forall i$. However, this can cause issues, namely the problem of overfitting. As illustrated in the polynomial models of Figure 2.4, a function that better approximates the dataset does not necessarily work well on data outside of it. Overfitting means memorizing solutions instead of finding patterns, meaning the generalization capabilities of a model are weak.

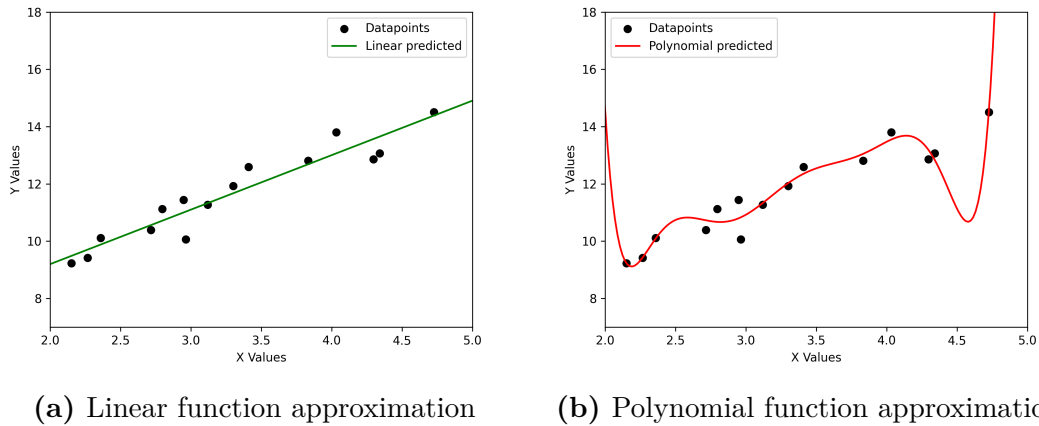


Figure 2.4: The overfitting problem

Therefore our goal is no longer to exactly cover $\mathcal{D}$, but rather to make descent guesses on the dataset, and beyond it. To do this we can separate a training and testing (or verification) dataset. We train only on the training set and backtest on the test set that our machine learning model has never seen before. If the test performance is worse

than the performance on the training set, we can be sure, that our model relied more on memory than pattern recognition.

To quantify how well our model did on a given input $\mathbf{x}$ we introduce the Loss metric $\mathcal{L}$. The loss function describes the quantity of error between our predicted output $f_\theta(\mathbf{x}) = \hat{\mathbf{y}}$ and the correct label $\mathbf{y}$.

An example of a common loss function is Mean Squared Error (MSE), where each index of the predicted and correct label is compared, squared, added together, and divided:

$$\mathcal{L}_{\text{MSE}}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \frac{1}{n} \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2 \quad (2.23)$$

In our application, the loss function most commonly used is Cross Entropy Loss. The output of a network is often a probability distribution, that we want to train to match a real probability distribution. To do this, we can calculate, how surprising a real event is, based on our predicted distribution, this average surprise is called Cross Entropy. We are least surprised when our predicted distribution is the same as the real underlying distribution. Therefore we want to minimize Cross Entropy for our model, and we can use it as a loss function.

$$\mathcal{L}_{\text{Cross-Entropy}}(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{i=1}^n y_i \cdot \log\left(\frac{1}{\hat{y}_i}\right) = \sum_{i=1}^n -y_i \cdot \log(\hat{y}_i) \quad (2.24)$$

Now that we have a good metric for what we want to minimize, we could describe our optimal parameters θ^* :

$$\theta^* = \arg \min_{\theta} \sum_{i=0}^N \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) \quad (2.25)$$

$$\theta^* = \arg \min_{\theta} \sum_{i=0}^N \mathcal{L}(f_\theta(\mathbf{x}), \mathbf{y}) \quad (2.26)$$

However, actually minimizing θ like this is quite a challenge. First of all, completely fitting the model to the data or overfitting, is bad for generalization. Second, the system of equations would be huge, on the order of the dataset size N times the output dimension, all while being non-linear. Another idea might be to sweep through the loss function with the parameters θ . However, this is also computationally impossible for DNN models, which have large amounts of parameters. Therefore we must use another method.

2.2.4 Stochastic Gradient Descent

We are unable to use information about the loss of our model globally, but we can use information about the local neighborhood of our current parameters. If we take the gradient of the loss function with regards to the parameters, we can find out which direction in parameter space corresponds with the quickest increase in loss, e.g. which

way is the function steepest upwards. To calculate the gradient, we simply take the partial derivative in each direction.

$$\nabla_{\theta} \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \theta_1} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial \theta_p} \end{bmatrix} \quad (2.27)$$

Figure 2.5 illustrates on a simple example function, that the opposite of the gradient is the quickest decrease.

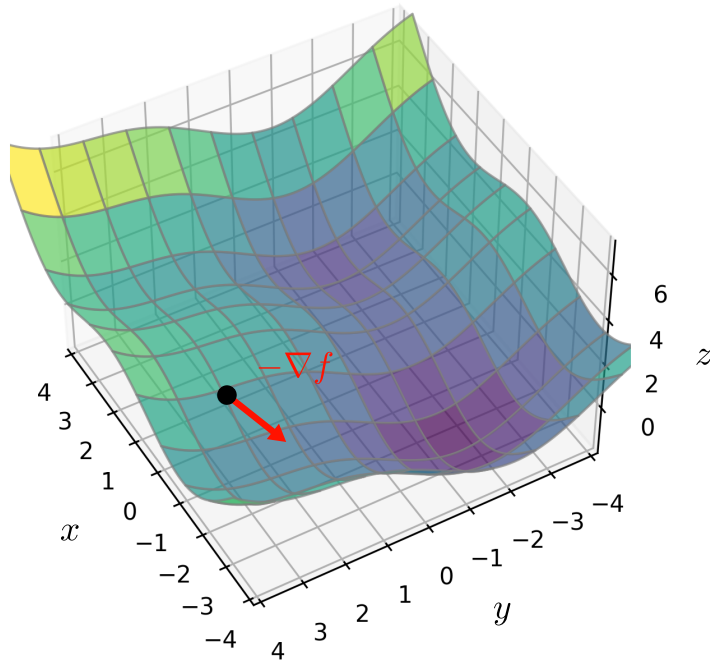


Figure 2.5: Gradient as the steepest direction, illustrated with a simple landscape

We can find the gradient of the loss function using the chain rule:

$$\nabla_{\theta} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \nabla_{\theta} \hat{\mathbf{y}} + \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \cdot \nabla_{\theta} \mathbf{y} \quad (2.28)$$

Since $\mathbf{y}$ isn't a function of θ , and $\hat{\mathbf{y}} = f_{\theta}(\mathbf{x})$, we can rewrite Equation 2.28:

$$\nabla_{\theta} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \nabla_{\theta} f_{\theta}(\mathbf{x}) \quad (2.29)$$

The method to compute the gradient of f is backpropagation. Because of the chain rule, we can build the gradient by calculating the derivatives at each graph point and then multiplying them together.

So the training process consists of taking steps, updating all of the parameters to best decrease the loss. Each step is made with the so-called learning rate (LR) μ .

$$\theta' = \theta - \mu \nabla_{\theta} \mathcal{L} \quad (2.30)$$

As we increase the learning rate μ , the training becomes faster as we take larger gradient steps. However a trade-off is made, we aren't able to find a precise minimum point, because we may accidentally jump over them. Figure 2.6 shows two paths of gradient descent to find the minimum of a simple function. In practice, this landscape is much more complicated.

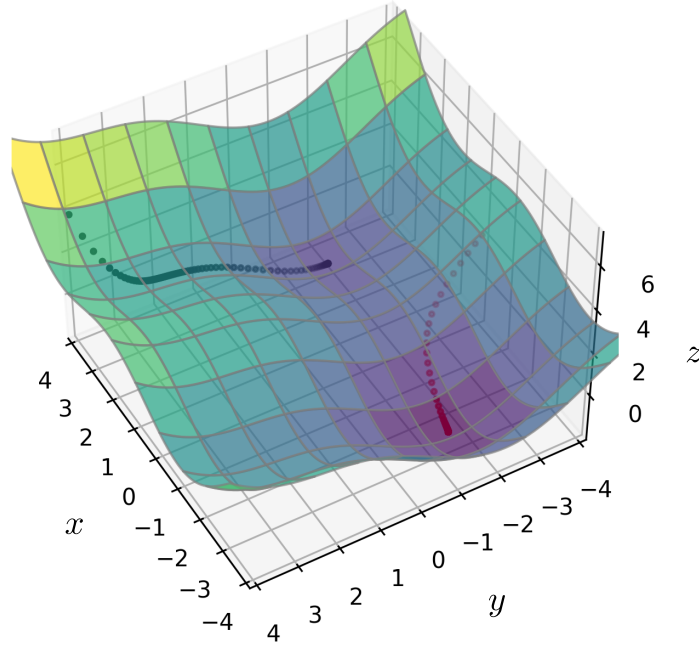


Figure 2.6: Gradient Descent on a simple loss landscape, initialized from two different starting parameters

It is clear that this method can not find the global minimum, highlighted by the fact that the two descents in Figure 2.6 converge onto different points. This, however, is not as big a problem as it might seem. Since the parameter space is huge, it is perfectly fine if we are able to find a good local minimum point.

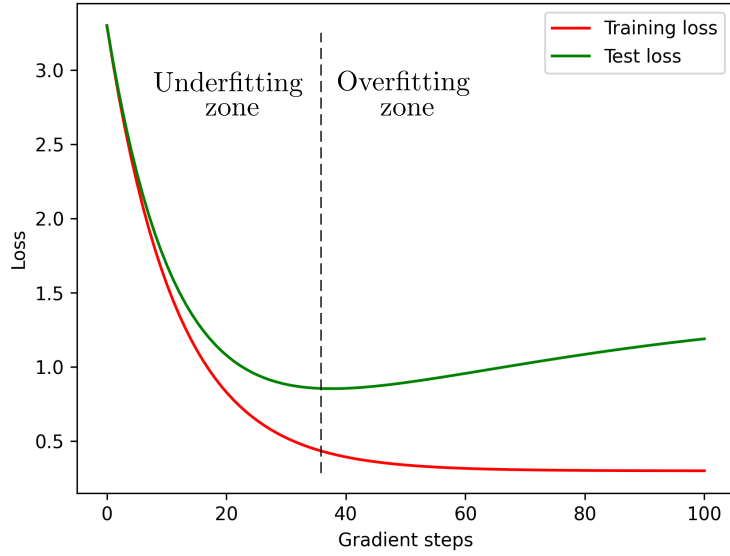


Figure 2.7: Loss decreasing with training steps. Training loss and test loss can be seen separating when overfitting begins.

In Figure 2.7 it is shown how the gradient descent algorithm changes the loss. When the training and test losses split off, our model started memorizing training data and begins to overfit. Since our goal is for the model to extrapolate well, and work on general data, the best model is found when the test loss is minimal.

In practice, deep learning is extremely effective at learning functions and extrapolating through pattern recognition. Additional techniques, such as Convolutional Neural Networks [23], Recurrent Neural Networks [24], and more have allowed it to solve difficult problems from within image recognition and natural language processing, to control engineering.

Within control engineering, two important areas have been studied: its ability to enhance the methods of traditional RL (Section 2.1), called Deep Reinforcement Learning (DRL) (Section 2.3), and the ability to learn from experts, called Imitation Learning (IL) (Section 2.2.5).

2.2.5 Imitation Learning

Imitation learning (IL) algorithms [25] have the same goal as any RL algorithm: complete a task as well as possible. However, the big difference between IL and RL is that, on the one hand, IL algorithms try to mimic a supposed expert as well as possible, on the other hand, RL algorithms try to maximize the long-term cumulative reward with respect to some value function. It is easy to see that vanilla imitation learning (also called behavioral cloning) is a special supervised learning problem, because the "correct" actions for a certain state are given. This way the performance of an imitation learning algorithm is bounded

by the quality of the expert demonstrations, while RL policies can (in theory) reach the optimal policy. Another downside of imitation learning algorithms is that they cannot generalize outside their training data, which makes them vulnerable to deviations in their environment (Figure 2.8). The main benefit of IL is that it produces a policy from already collected data, and because the labels are given it is much more sample-efficient than RL.

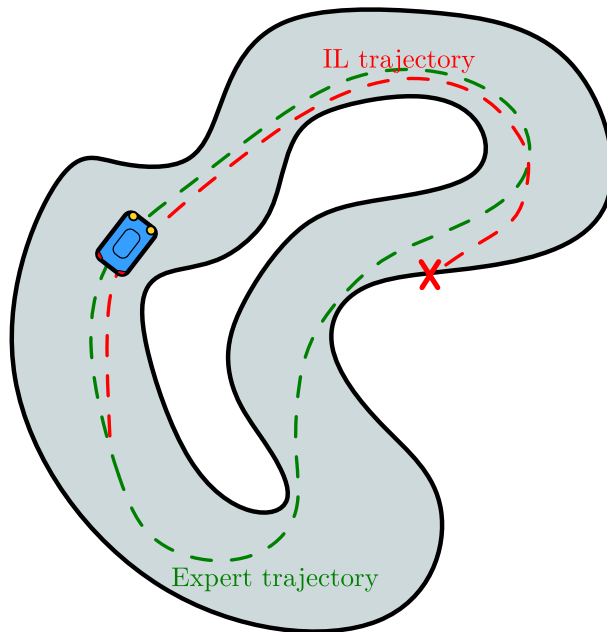


Figure 2.8: A small deviation from expert trajectories leads to unrecoverable states when using behavioral cloning

2.3 Deep Reinforcement Learning

The curse of dimensionality (Section 2.1.4) severely limits the usability of RL to strictly low dimensional state and action space problems. Deep Reinforcement Learning is the use of function approximation using deep neural networks within RL, mainly with the goal of reducing the storage needed for memorizing all states by learning the patterns that govern the problem. In practice, this means we represent functions we otherwise would have used tables for using neural nets, such as value, action-value, and policy functions.

This has proven to immensely increase the problems to which reinforcement learning is applicable, many difficult problems have been solved using DRL in the last few years. One of the most widely used examples is ChatGPT, which was fine-tuned using Reinforcement Learning from Human Feedback (RLHF) [26], drastically improving its usability.

In general, the two main methods of DeepRL are the Deep Q-Networks (DQN) and the Policy Gradient method. An introduction is provided into how these methods are used, and their relevant characteristics. It is important to note, that these methods, unlike the tabular reinforcement learning presented in Section 2.1, do not converge to globally optimal solutions, but rather find approximate locally optimal solutions.

2.3.1 Deep Q-Networks

The traditional method of Q-Learning (see Section 2.1.2) uses a lookup table to store the values of the action-value function Q . This means the required number of stored elements is $|\mathcal{S} \times \mathcal{A}|$, therefore both must be finite, so continuous spaces must be discretized.

In 2013, a groundbreaking method was created by researchers at DeepMind. The method, called Deep Q-Networks [27] was used to train an agent to play on the Atari 2600, successfully playing each game.

A Deep Q-Network is a neural network constructed to approximate the values of the Q function. The input of the network is the state, and it has an output for each possible action, which specifies the action-value function for the given action. So Q_θ denotes the approximation of the Q function with parameters θ .

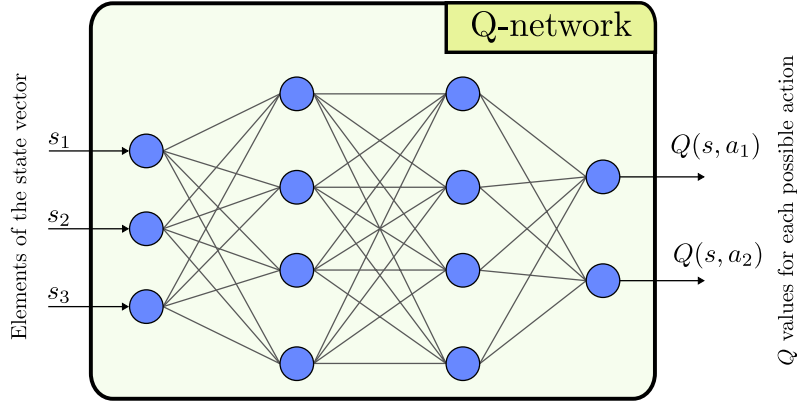


Figure 2.9: Q-Network structure

Therefore the greedy policy can be expressed simply:

$$\pi(s) = \arg \max_a Q_\theta(s) \quad (2.31)$$

The loss of the Q-Network is its deviation from the Bellmann equation for the action value function (see 2.5). The action-value function returns our expected discounted cumulative reward if we choose a given action in a given state, and then continue on by always choosing our next action based on the greedy policy π from equation 2.31. This means during training, when we are taking trajectory $\tau = (s_0, a_0, s_1, a_1, \dots)$, we can calculate our loss for each timestep t :

$$y_t = R_t + \max_{a'} Q_\theta(s_{t+1}, a') \quad (2.32)$$

$$\mathcal{L}(\theta) = \sum_{t=0}^{\infty} (y_t - Q_\theta(s_t, a_t))^2 \quad (2.33)$$

Thus using gradient descent, we can find a Q-network that satisfies the Bellmann equations, therefore finds a good solution to our optimization problem. This method allows

for a continuous state space, without needing large amounts of storage. As the Atari 2600 demonstration shows, it is much more effective and widely applicable than the previous tabular Q-learning method.

2.3.2 Policy Gradient

In policy gradient methods a policy is directly learned in the form of a neural network, without the approximation of a value function. The policy is stochastic, meaning it chooses any action with a given probability: $\pi_\theta(s) = [p(A_1|s), p(A_2|s), \dots]^T$. For any trajectory $\tau = (s_0, a_0, s_1, a_1, \dots)$, the probability of occurring when using policy π_θ , initial state distribution $p(s)$, and transition function $p(s_{t+1}|a_t, s_t)$:

$$p(\tau|\theta) = p(s_0) \prod_{t=0}^{\infty} \pi_\theta(a_t|s_t) \cdot p(s_{t+1}|a_t, s_t) \quad (2.34)$$

Our expected finite-horizon undiscounted return (the expected sum of rewards $R(\tau)$), when sampling τ trajectory from the environment and policy is:

$$R(\tau) = \sum_{t=0}^T R_t \quad (2.35)$$

$$J(\theta) = \sum_{\tau} p(\tau|\theta) R(\tau) \quad (2.36)$$

Therefore the gradient can be calculated. Because of the MDP properties, the reward only depends on the trajectory.

$$\nabla_{\theta} J(\theta) = \sum_{\tau} R(\tau) \cdot \nabla_{\theta} p(\tau|\theta) \quad (2.37)$$

Because of the rule for the derivative of logarithms $(\ln(x))' = \frac{x'}{x}$:

$$\nabla_{\theta} J(\theta) = \sum_{\tau} R(\tau) \cdot p(\tau|\theta) \cdot \nabla_{\theta} \ln(p(\tau|\theta)) \quad (2.38)$$

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim \pi} [R(\tau) \cdot \nabla_{\theta} \ln(p(\tau|\theta))] \quad (2.39)$$

Since the logarithm of a product is a sum, based on 2.34:

$$\ln(p(\tau|\theta)) = \sum_{t=0}^T \ln(\pi(a_t|s_t)) + \sum_{t=0}^T \ln(p(s_{t+1}|a_t, s_t)) + \ln(p(s_0)) \quad (2.40)$$

We know the transition function and initial state distribution do not depend on the model, or its parameters. Therefore we can get back our reward gradient (the negative of which would be our loss):

$$\nabla_{\theta} \ln(p(\tau|\theta)) = \sum_{t=0}^T \nabla_{\theta} \ln(\pi(a_t|s_t)) + 0 \quad (2.41)$$

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim \pi} \left[R(\tau) \cdot \sum_{t=0}^T \nabla_{\theta} \ln(\pi(a_t|s_t)) \right] \quad (2.42)$$

Since getting the log-probs of our stochastic policy network is simple, we are able to calculate our reward for trajectories the agent takes during training, and update our parameters using any gradient descent-based optimizer (for example Adam [28]), this time maximizing our cumulative reward directly.

$$\theta' = \theta + \mu \cdot \nabla_{\theta} J(\theta) \quad (2.43)$$

The presented derivation is for the Vanilla Policy Gradient Theorem. Unfortunately using only this theorem to create RL agents results in rather unstable policies, which are really hard to train. The vanilla policy gradient algorithm has been improved by using methods such as gradient clipping or adding value approximation to the system for additional stability. Modern algorithms like DDPG [29], PPO [30], A3C [31] are versions of the policy gradient, usually incorporating value-based elements as well. These algorithms are much easier to train and converge faster than pure value-based algorithms, which makes them a common choice for RL practitioners. It is useful to note, that these algorithms are regarded as on-policy methods, this means the agent is only able to learn from decisions sampled using its policy.

2.4 The sim2real problem

Deep reinforcement learning techniques require a lot of training samples (often millions) to fully solve a problem, but it is most of the time quite difficult to gather all the data in the real scenario, so practitioners use simulators to train their policies. But training an RL agent in a simulator is not such a simple task as it seems, because perfect environment models do not exist and there are always discrepancies between simulated and real scenarios. As discussed earlier, learning-based methods are quite sensitive to the training data distribution, and reinforcement learning can also overfit to a specific reward or environment. Combining these two techniques in deep reinforcement learning just makes this issue much more severe and policies that seem to be working well in the simulator are often unusable in the real world. In other applications, the agent learns to specifically exploit the weaknesses of the simulator that do not exist in the real environment, e.g., in paper [32] the agents learned to exploit the physics of the ramp element. The sim-to-real gap is quite common when applying learning-based methods on control tasks, a good example is [33], where the researchers use simulated pure-pursuit experts to train policies using imitation learning. On both the learning track and on different general simulated tracks, their best IL policy is able to complete laps comparably well to the expert, possibly as quickly as 8.0m/s. However, when evaluated in a real test environment only one policy can complete a lap at 3.0m/s, all while wobbling around, indicating a strong loss of capability during the sim-to-real transfer. The authors say that it might be the result of the change in friction between the simulated and real environment. The problem of transferring the simulator-learned knowledge into other domains (real-world) is called the sim-to-real (sim2real) gap. The successful bridging of the sim2real gap depends greatly on the generalization ability of the policy.

Fortunately, there are some solutions to this problem, that allow agents to be trained inside a simulator and then deployed in the real environment. The first solution idea is domain randomization (Figure 2.10, a technique that perturbs environment parameters (for example friction, lighting, color palette) wildly during training, thus allowing the agent to see a broader range of scenarios, achieving a more robust policy. The second solution is a technique called learning with disturbances. As the name suggests this method introduces disturbances (randomly spawning objects) to the training environment. The idea is the same as in domain randomization, the agent sees a broader data distribution during training and becomes more robust. Also, the imperfection of the environment model can be seen as a kind of distraction, and the agent learns to cope with distractions, so it learns to cope with the environment model imperfections in the process. The third possible solution is observation preprocessing, where the observation of the policy is altered in some way that makes transferring skills in the simulator to the real world easier. One idea here is to make the observations lower-dimension and more high-level to achieve a more general agent, that can easily transfer its knowledge into a real scenario. Another possible solution is to create agents that are much more adaptive than vanilla RL agents and this way make them transferable to real situations. This type of solution will be explored in Section 7.1.

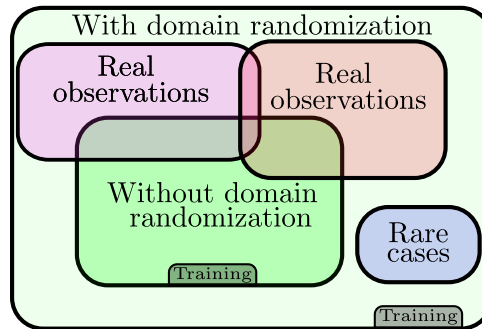


Figure 2.10: Data distribution when using domain randomization

2.5 Open-source platforms

In this work, we chose Python as our programming language, because of its versatility, ease of use, and great library support for reinforcement learning algorithms. The two major libraries that we used were: `stable-baselines3` [34] and `gymnasium` [35]. The F1TENTH platform [1] is also open-source, anyone can build one, given the parts.

2.5.1 `gymnasium` module

The `gymnasium` module plays a crucial role in modern reinforcement learning development and research because it standardizes the whole RL formulation. With the use of this library the state space, action space, and the whole agent-environment interaction can

be defined parametrically and this way the implemented RL algorithms are easily swappable. In the module, a bunch of built-in environments are found, on which researchers can benchmark their algorithms (Figure 2.11). The module also allows creating custom environments, for which the previously implemented RL algorithms also work. The `gymnasium` environments (even the custom ones) offer easy parallelization for training.

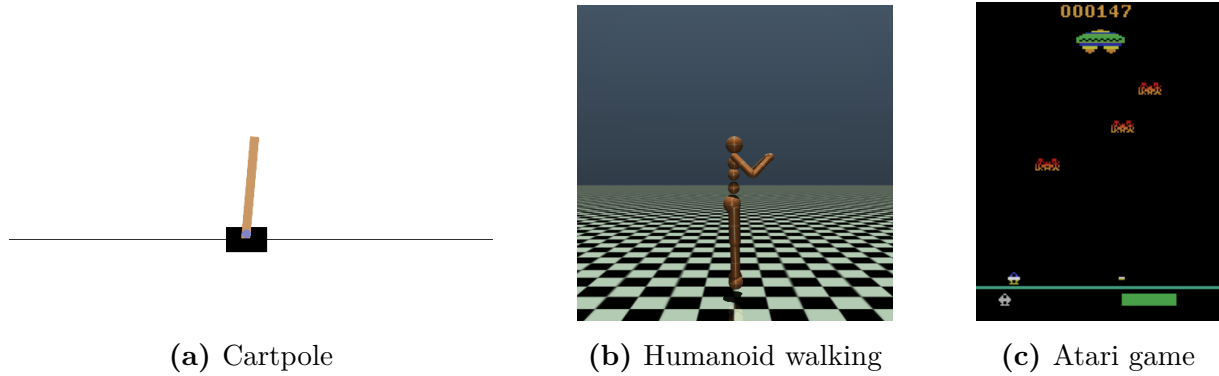


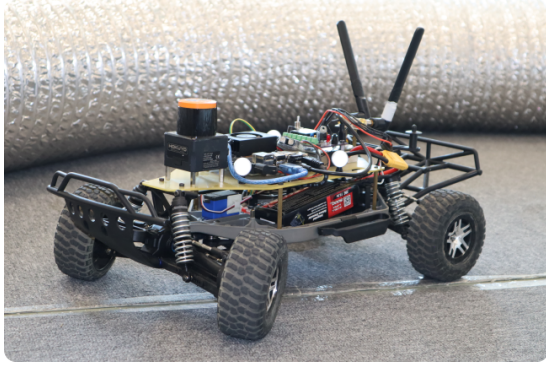
Figure 2.11: A few built-in environments

2.5.2 `stable-baselines3` module

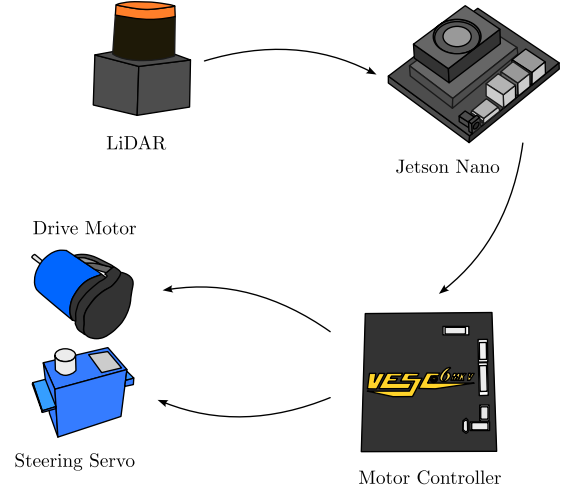
The `stablebaselines3` module has all the major, modern RL algorithms implemented. The implementations are compatible with `gymnasium` environments, so it is easy to try out different algorithms on the same problem. The project is completely open-source, and all the code is available on GitHub. PyTorch [36] is used as the deep learning library behind the algorithms. Fortunately, all the algorithms come with reasonable hyperparameters, which makes the library almost plug-and-play.

2.5.3 F1TENTH platform

The open-source F1TENTH platform was created to boost research in the area of autonomous racing and self-driving cars in general because it is a relatively low-cost testing platform for researchers. The car is based on a high-power Traxxas RC car. The vehicle is equipped with a 2D LiDAR sensor that scans the environment at an angle of 270 degrees, returning an array of 1080 distance measurements with a sampling rate of 40 Hz. An IMU sensor is also located on the car. In F1TENTH competitions the teams must do all computations on the onboard Jetson Nano, which has limited computing power. The platform developers provided a ROS package, which makes working with the hardware easier because on the `scan` topic one can receive the LiDAR scans and on the `drive` topic one can send control signals by specifying a target speed and a turning angle.



(a) Our F1TENTH car



(b) Control scheme of the car

Figure 2.12: The F1TENTH platform

2.6 Dynamics model

The dynamics model used during this research was a non-linear bicycle model, taken from [37], which can handle slipping and has a nonlinear tire model, but the center of mass shift, caused by accelerations, is not dealt with. Figure 2.13 shows its structure. When calculating the dynamics it is assumed that tire forces are perpendicular to the plane of the tire.

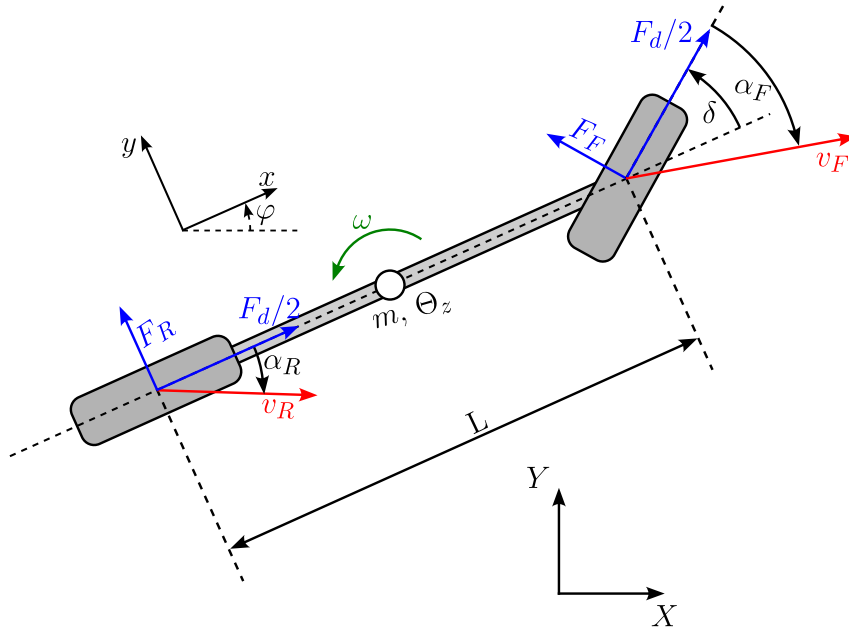


Figure 2.13: Bicycle model

By writing out Newton's II. in the local coordinate system and approximating $\cos(\delta) \approx 1$

when projecting $F_d/2$ on the front wheel, we get:

$$ma_x = -F_F \sin(\delta) + F_d \quad (2.44)$$

$$ma_y = F_F \cos(\delta) + F_R \quad (2.45)$$

$$\Theta_z \dot{\omega} = (F_F \cos(\delta) - F_R) \cdot \frac{L}{2} \quad (2.46)$$

Getting the acceleration in a system that rotates with ω angular velocity, is done by taking the time derivative of the $\vec{v}$ speed vector.

$$\mathbf{a} = \dot{\mathbf{v}} + \omega \times \mathbf{v} = \begin{bmatrix} \dot{v}_x - \omega v_y \\ \dot{v}_y + \omega v_x \\ 0 \end{bmatrix} \quad (2.47)$$

Substituting Equation 2.47 back to Newton II. results in the following:

$$m(\dot{v}_x - \omega v_y) = -F_F \sin(\delta) + F_d \quad (2.48)$$

$$m(\dot{v}_y + \omega v_x) = F_F \cos(\delta) + F_R \quad (2.49)$$

$$\Theta_z \dot{\omega} = (F_F \cos(\delta) - F_R) \cdot \frac{L}{2} \quad (2.50)$$

Based on this, the slip angles (the angle between the longitudinal axis of the wheel and its velocity vector) for the front and rear tires can be calculated as follows:

$$\alpha_F = \arctan\left(\frac{v_y + \omega \cdot \frac{L}{2}}{v_x}\right) - \delta \quad (2.51)$$

$$\alpha_R = \arctan\left(\frac{v_y - \omega \cdot \frac{L}{2}}{v_x}\right) \quad (2.52)$$

The tire forces F_F , F_R , which are functions of the slip angles for the respective tires and four constants, are calculated using Pacejka's nonlinear Magic formula [38]. The constants for the formula can be found in the Appendix. Using the tire forces the accelerations can be calculated, which helps to approximate the velocity of the vehicle with the Forward Euler method.

3 Our contribution

3.1 Development of the simulator

To enable the training of RL agents for our chosen F1TENTH platform, we needed to use a simulator. Several simulators with accurate dynamics models are available for our specific vehicle, such as the ROS and Gazebo-based [39], and the AutoDRIVE-based [40]

environments. However, these have some weaknesses, such as their lack of customizability in terms of both the dynamics and the environment, lack of support for parallelization, and compute inefficiency.

In contrast to this, we wanted to create a lightweight simulator fully integrated fully Python as a Gymnasium environment (more in 2.5.1), enabling rapid prototyping and total customizability. Because our simulator is based on the Gymnasium environment API, it also supports parallel training, which was of great use to us.

Our simulator was created in an object-oriented manner, making the adjustment and addition of simulation and model elements very simple. These features were used extensively for testing and varying many different ideas quickly such as the observation creation, the environment randomization, and the disturbance engine (to be discussed in 3.2). The dynamics model in our simulator was from [37], which is discussed in 2.6.

As the reinforcement learning problem is formulated in terms of an MDP, our simulator needs to implement a starting state distribution (resetting the environment), and a transition function (getting the next observation and reward from the current state and chosen action).

3.1.1 Tracks

It is common for F1TENTH teams to test their control schemes only on Formula 1 racing tracks, however, we found that these are sometimes not representative of the ratios of real F1TENTH tracks. Therefore we created a tool for drawing, editing, and saving custom tracks, for us to use. The edges of the tracks are built up in loops. Similarly, a centerline

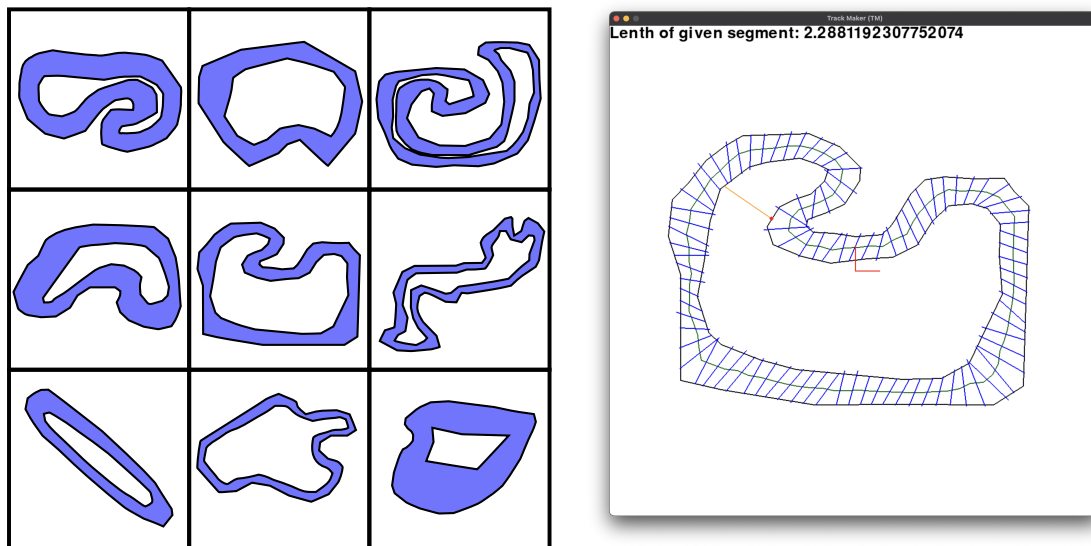


Figure 3.1: Our training tracks and track creation tool

can be defined, for possible use during spawning and to provide additional information about runs. Furthermore, the user is able to define checkpoints, which can be used in

rewarding and testing. The tool also enables the use of background images for reference, viewport scaling, and precise track size adjustment.

3.1.2 Getting the starting state

Whenever a new episode is started, a state is randomly sampled from the initial state distribution. In Gymnasium environment terminology this function is called `reset()`. Practically, this means selecting a point on the centerline, and pointing the car roughly in the right direction (randomly clockwise or counter-clockwise). Upon a new episode, the simulator currently has support for varying tracks, track sizes, friction coefficients, action delay, control frequency, and placing random gaps into the walls. All these techniques help to achieve better generalization abilities.

3.1.3 Transition function

The transition function (sometimes named transfer function) within the MDP defines what happens when selecting a given action in a given state. In the Gymnasium environment class, from which we derive ours, the transfer function is called `step()`.

Algorithm 2 Transition function

```

Input: Action a
Generate  $(v, \varphi)$  speed, turn pair from  $a$ 
Sample the sampling period  $\Delta t$ 
Sample the action delay  $\Delta t_a$ 
Calculate the state of the car with dynamics for  $\Delta t_a$  based on the previous action  $\mathbf{a}_{t-1}$ 
Apply the smoothing and limiting functions on  $(v, \varphi)$ 
Calculate the state of the car for  $\Delta t - \Delta t_a$  with the control signal  $(v, \varphi)$ 
if Stopped early then
    done=true
end if
Calculate the position of moving obstacles for time  $\Delta t$ 
Count checkpoint crosses
if Car collided then
    done=true
end if
Calculate the reward  $r$ 
Generate LiDAR reading l
Append l and  $(v, \varphi)$  to the observation buffer  $\mathbf{b}_o$ 
Select the last 5 elements of  $\mathbf{b}_o$  to observation  $o$ 
return (o,  $r$ , done)

```

3.1.4 Control signal

Because we wanted the simulator to support both discrete and continuous action spaces, the action is first converted to a control signal. This, however still leaves room for harsh changes in speed and steering. To alleviate this we smoothed the input by limiting its rate of change, allowing for a more realistic simulator.

3.1.5 Checkpoint and collision detection

Checkpoint crosses are registered by taking the current and previous positions, and checking for intersection. The direction of the cross is also kept track of, to help when an agent tries to farm rewards by moving back and forth over a checkpoint.

Collisions were also originally calculated using segment intersections. However, we found checking for LiDAR points inside the car made for a much faster and perfectly reliable method.

3.1.6 Generating the LiDAR reading

During the development of the simulator, the algorithm was changed multiple times. The first version was a naive brute force algorithm that checked rays against all segments with complexity $\mathcal{O}(n_r n_s)$, where $n_r \leq 1080$ and $n_s \approx 10^2$ are the number of rays and the number of segments respectively. To train an agent in our setting, at least around 10^6 steps are required, which with the first method needed multiple days to converge. We wanted to improve on this, as the simulator was running slower than real-time. For the next iteration, we implemented the intersection check in C which significantly improved on the Python version, more than doubling the speed.

Finally, we implemented a dynamic programming method, that checks only so-called active segments for intersections with a certain ray. Segments of the wall are activated and deactivated based on the polar coordinates of their endpoints in the car system. In our case this means that on average 2 segments must be checked for intersection, which greatly speeds up the training, allowing for 10x real-time on a single CPU.

3.2 Development of the RL training stack

3.2.1 Our problem in terms of RL

Any solution to the problem of autonomous racing requires an agent to be able to make informed decisions every moment. Since there does not exist a "correct solution" to every situation, we decided that the best approach to this problem is RL: the agents can learn about their environment, and optimize in light of previous experiences. To use well-known RL algorithms, the formulation of the problem as a POMDP was needed. Every couple

Algorithm 3 Novel LiDAR Generation Algorithm

Input: track_segments, dynamic_objects, ignored_IDs, max_range, beam_parameters
Output: scan array
Initialize on_list and off_list as empty lists
Initialize active_seg and active_dyn as empty sets
Initialize scan array with max_range
for each track_segment **do**
 Calculate start_angle and end_angle, account for wrap-around if necessary
 Append segment points to on_list and off_list
end for
for each dynamic_object **do**
 if object is not ignored and is spawned **then**
 Calculate start_angle and end_angle, account for wrap-around if necessary
 Append segment points to on_list and off_list
 end if
end for
Sort on_list and off_list by angle
for each beam_direction **do**
 Update active_seg and active_dyn using on_list and off_list
 Calculate minimum distance for the beam using active_seg and active_dyn
 Update scan array with calculated distance
end for
return scan

of milliseconds, a new LiDAR scan is received. Based on this scan (augmented with a bit of extra information from the previous timesteps) the RL policy decides what low-level action to take i.e. it controls the speed and turn angle of the car. The details are discussed in Section 3.2.2.

To reach a successful policy, what good behavior is needs to be properly defined. We wanted to create an agent that can go quickly and safely on a racetrack with mostly continuous walls. Defining a reward that teaches this objective to the agent is a tricky task, so we tried many versions that are discussed in Section 3.2.3.

All training in the simulator is done in episodes, which end when the agent collides, stops moving for a longer period, or the predefined time horizon is reached. The details of the training scheme are discussed in Section 3.2.3.

This section covers the main challenges and solutions that enable the model to learn to race effectively in the simulator. The decisions we made in terms of reward, observation and action structure, and training schedule are explained and contrasted with other commonly used methods we tried.

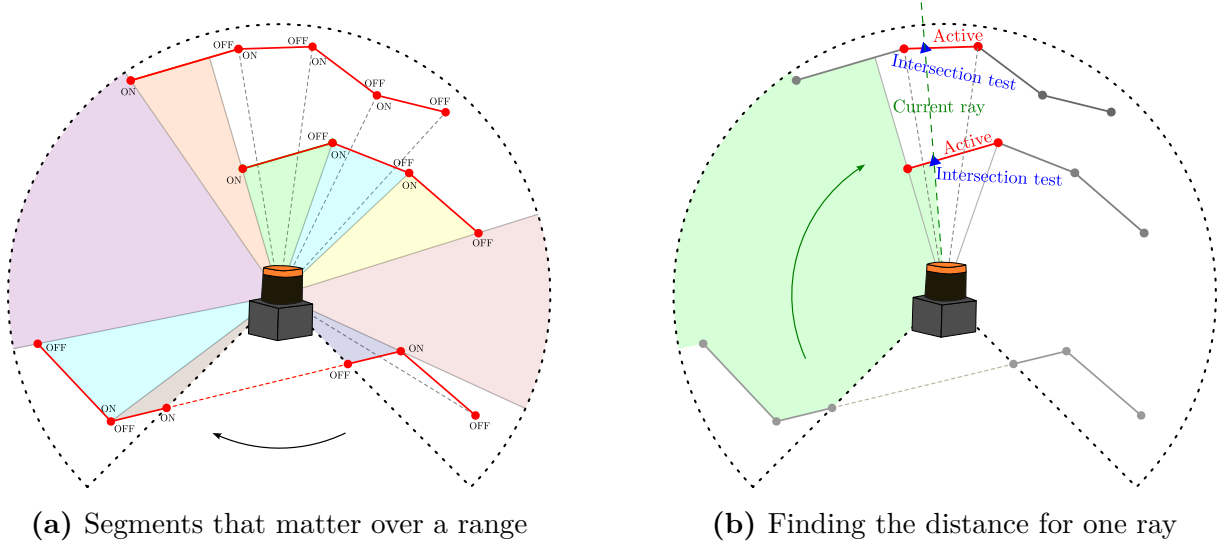


Figure 3.2: Figure showing how the segments are used in the algorithm

3.2.2 Model structure

The first intuitive method that popped into our head is to have a policy network that directly takes the full LiDAR scan as an observation and outputs a continuous-valued reference speed (even negative) and steering command every Δt time step, directly sent to the motor driver. However, using this decision-making structure results in weak agents that suffer from the following problems:

- **Inability to learn:** This simple type model was difficult to train, rarely converged onto any useful solution, and even the best models were quite weak, compared to later iterations. It was a common occurrence, that models would not start moving and would stay in one place for the whole episode.
- **Stuck in loops:** Another common problem was, that agents would get stuck in a position (such as a corner), moving back and forth without any sign of getting out. We hypothesized this to be caused by the lack of short-term memory, the model does not remember that it just tried to exit a dead end, and tries to go back in.
- **Uninformed speed level:** These models were almost completely unable to take sharp turns without crashing because there was no information about the current speed in their observation (containing only the LiDAR scan). They often crashed into walls without any apparent reason.
- **Slow reaction times:** The LiDAR is in theory sampled at a constant rate of 40Hz, however in practice this can vary quite a lot on the car. Because of this, the original concept was to take the last received scan every Δt time, and make a decision then. With values of $\Delta t = 0.1s, 0.05s, 0.025s$ the testing of our methods showed the agents had a slow reaction time, because of the lack of synchronization between the observation and decision-making.

As a solution, we employed new methods targeting the specific problems:

- **Multiple LiDAR scans:** To give the policy more information about its state, we created an observation buffer that contains previous LiDAR scans. Our thought process was that this would help with the awareness of the policy, stopping the car from getting stuck in loops, and giving it more information about how fast and where it is going.
- **LiDAR synced decision making:** A new method was employed, where a decision is made directly after a scan is received. To counteract the variability of this new dynamic timestep $\Delta t'$, the policy network is also supplied with this temporal information.
- **Limited action space:** The ability to go backwards was removed. In the real race situation, there is never a need for backing up. This way the car was forced to make a good decision the first time.
- **Discrete action space:** We employed discrete actions to reduce the action space of the model. This helps because of improved exploration, and easier credit assignment (Section 2.1.6). Policies with discrete action spaces are far easier to train than continuous ones.
- **Reduced observation space:** The curse of dimensionality (Section 2.1.4) makes training difficult, so we employed a custom state aggregator type observation pre-processing, which cut the size of our observations by almost 80%. This method helped with faster convergence and better generalization.

The final model structure can be seen in Figure 3.3. Additionally, we tried some other methods, that were not as good as the final version.

- **Only steering:** To counteract getting stuck in one spot, we hypothesized that forcing the car forward would force the policy to at least make good steering decisions. To prevent slip, the speed command was chosen based on the turn command using the equation $v = v_{max} \cdot \cos^3(\varphi)$. Even after attempted tuning this model was not deemed capable.
- **Overseer:** To stop the problem of models hitting walls and obstacles without apparent reason, we created an overseer algorithm method. Upon the detection of a LiDAR point within a given range in front of the car, the overseer would take over and stop the car, back up a bit, and give back control. The problem was, that with too small a breaking distance the car would crash regardless of braking, and a too large distance would prevent making turns altogether. Safe driving requires a stronger ability to determine what situation is dangerous.

- **Finetuning a model initialized with imitation learning:** There was a plan to train the RL algorithm in the real world, but vanilla PPO is way too data-inefficient for doing that. To overcome this problem, we tried to first train the policy neural network with our driving data (providing expert demonstrations) using supervised learning techniques, and then train the network further with RL. Unfortunately, we did not manage to make this technique work, mainly because of the lack of training data and our inability to drive the car fast around the track.

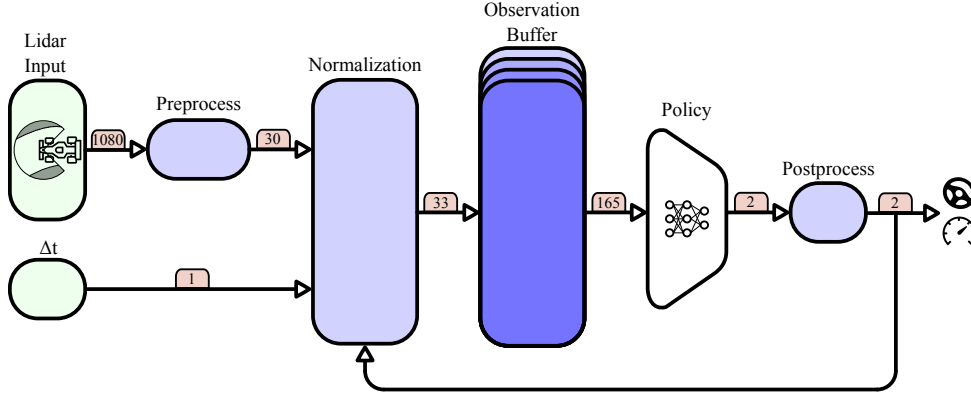


Figure 3.3: Structure of the model

3.2.3 Reward mechanisms and training scheme

To create an efficient policy, a good reward function needs to be found. The first idea that intuitively comes to mind: the reward should be $-t_{lap}$. However, with this approach learning is extremely difficult, because of the sparse reward (more in Section 2.1.5). It is hard to create an exploration strategy that finishes even a single lap before ever receiving any reward, halting the learning process entirely.

Iterating on this we first tried giving rewards based on how fast the car was going, punishing crashes. This method allowed the agent to get the hang of racing quickly because it was given information every single timestep. This scheme resulted in agents that were able to do decently but were reckless and did not plan for the future, often entering turns quickly. Increasing the crash penalty either didn't help significantly in the credit assignment problem or caused the agent to fear even the smallest movements, staying in the spawn position for the whole episode. After a lot of testing, we added a new element to the reward function: checkpoints. This way the agent is encouraged to make progress along the track. After trying out multiple weights for the components of the reward, we found that an effective reward function $r(t)$ has to punish crashes, reward checkpoint passes, and improve speed, as follows:

$$r(t) = \alpha_1 v(t) + \alpha_2 |C(t)| + \alpha_3 T_{cr} - \alpha_4 T_{coll}, \quad (3.1)$$

where

- t denotes the actual time step, and α_i , $i = [1; 4]$ represent scalar weights for scaling the elements in the reward,
- $v(t)$ is the vehicle speed (determined from the previous control output),
- $C(t)$ is the turning angle,
- T_{cr} is a binary indicator (1 if a checkpoint is crossed, 0 otherwise),
- T_{coll} is a binary indicator (1 if a collision occurs, 0 otherwise).

The agents were trained inside the simulator and then deployed without any fine-tuning into the real world. From the algorithms implemented in the `stable-baselines3` library, we chose PPO for our RL algorithm because it is easy to train and sample-efficient enough. An Intel Core i9-12900K processor was used for training, running 48 environments in parallel. This modern CPU allowed for 10x real-time training. Because exploration is built into the PPO algorithm, we did not have to worry about the exploration-exploitation trade-off (more in Section 2.1.3). All the hyperparameters used for the training are in the Appendix. The γ discount factor (Section 2.1) parameter is of particular importance because if γ is bigger the agent will give more weight to future rewards, causing it to become more forward-looking.

3.3 Bridging the sim-to-real gap

One of the downsides of reinforcement learning is its data hungriness, i.e. learning algorithms need many (in practice often millions of) interactions with the environment before finding a solution. To address this issue most practitioners use simulators for training, but this often leads to a policy specialized to only the simulated surroundings, that cannot perform in the real environment. Sometimes the difference between the two domains becomes large enough for the policy to become unusable.

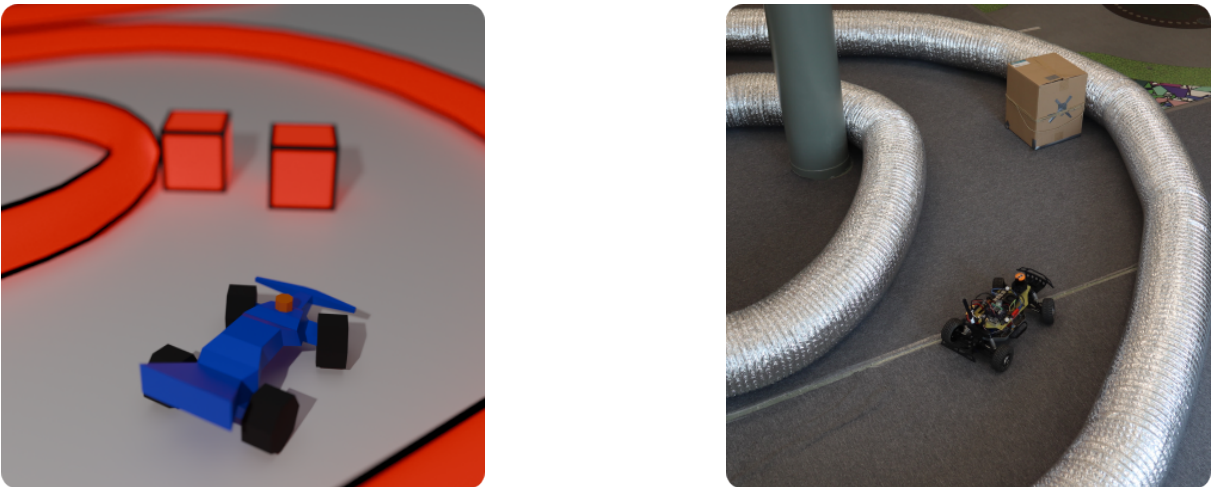


Figure 3.4: Illustration of the sim-to-real problem in autonomous racing

Our proposed sim-to-real transfer framework is illustrated with Figure 3.5, where all the elements of the method can be seen.

- **Friction perturbation:** To create a robust policy that can perform in almost all friction conditions, we varied the friction during training in a wide range.
- **Learning with disturbances:** Another important aspect was the safety of the resulting policy, and thus, we extended the simulator by adding obstacles (both moving and static) in front of the car at random times. This way the agent learns to fear the uncertainty in sharp corners and reacts to sudden unexpected situations, driving safer compared to agents that have not seen any obstacles at all. Additionally, this ability to avoid moving obstacles also translates into the ability to overtake the opponents in the context of head-to-head racing¹.
- **State aggregation:** From work [13] we concluded, that using state-aggregation methods is a simple form of generalization. In practice this means we split the LiDAR beams into 30 sections scaled thinner in the front and wider on the sides, selecting the smallest value from each section, and feeding each value into our network. We hypothesized that it could improve the sim-to-real transfer abilities of the models through reducing reliance on the precise modeling of the simulated LiDAR observations, also reducing the dimensionality of the problem while retaining important information.



Figure 3.5: Illustration of our sim-to-real transfer framework

4 Experimental results

To measure the effectiveness of our proposed method, we conducted an ablation study: we performed tests in both simulated and real environments with and without different

¹See <https://youtu.be/1P7m1awgVDo> for examples

elements of the training stack and model architecture. For comparison, we chose to use the disparity extender method [6], which has proven highly effective, easy to implement and tune, and has won many F1TENTH competitions.

Testing in simulation provides opportunities for creating very controlled experiments and gathering large amounts of data for inference. In addition to this, the real environment provides good testing grounds for robustness and sim-to-real transfer. To combine the two we designed three tracks in the real world and modeled them inside our simulator. None of the tracks were seen by the models before testing.

Before the tests were conducted, we observed that the models in the more simple simulated environments got higher average rewards during training. We hypothesized this strong behavior to be specific to the environments these models learned in, and thus would not correlate with policy performance in more challenging situations.

4.1 Tested policies

- **Baseline:** All of the methods investigated are used: The simulator uses full domain randomization (friction, control frequency, and track scale are perturbed), and contains disturbances in the form of obstacle spawning, the observation preprocessing method is also applied. The policy has a discrete action space.
- **Continuous:** Identical to the baseline policy, except it has a continuous action space.
- **Fixed friction:** The friction is fixed during the training process.
- **No obstacles:** There are neither dynamic nor static obstacles present in the training experience of this agent.
- **No preprocessing:** The preprocessing step is removed from the model structure, but all the other elements and domain randomization still remain.
- **Disparity extender [6]:** The reactive control scheme used for comparison with the RL models. The car targets the furthest point on the LiDAR scan that it deems reachable without hitting a wall. Every time a new scan is received, a new point is targeted, allowing the car to move forward along the path, without hitting the walls. The algorithm was tuned to the best of our ability to provide a strong comparison to the RL models.

Because of the curse of dimensionality (Section 2.1.4), the continuous and unprocessed observational models required more time to converge, but we made sure that all models got as good as they could inside the simulator before testing. This meant training all of them for 15M steps and choosing the best version for each, these remained static in the future. No training was done outside the simulator, the agents were transferred zero-shot into the real domain.

4.2 Test tracks

During testing, the tracks used were similar in shape and size to those used during training, but the agents had not seen them prior to the experiments. The tracks were created to represent hard problems both geometrically and in terms of sim-to-real transfer, and are ordered by difficulty. The real-life track setups can be seen in Figure 4.1.

- **Track 1** is a loop on a carpet, designed to have sudden turns that require the agent to maneuver around them. The carpet has a high enough frictional coefficient for the racecar to not slip at any point during the testing, thus this test only measures the model’s ability to resolve dynamics model inaccuracies such as vehicle size, wheel position, and motor specifications. All of these parameters are approximately correct inside the simulator, however were never specifically tuned. The models are tested with a maximal speed of 5m/s.
- **Track 2** is a simple donut shape situated on a slippery epoxy-coated flooring. This test was designed to measure the quality of the policies over the grip limit. The models are tested with a maximal speed of 4m/s.
- **Track 3** follows a more complex banana shape, that requires the vehicle to arrive into sharp corners prepared and make turns calculating for loss of grip and uncertainties in the subsequent path. The models are tested with a maximal speed of 4m/s.

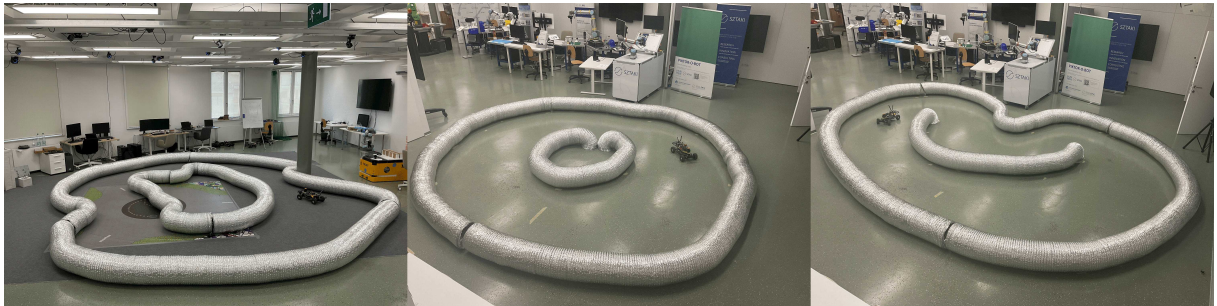


Figure 4.1: Tracks 1-3 from left to right

4.3 Simulator testing

Inside the simulator, we had the policies complete laps around the tracks. Every timestep the state data was recorded, creating a dataset containing many trajectories. This data enables analysis of the models in terms of speed profiles, and risk taking.

As can be seen in Figure 4.3, the baseline, continuous, and disparity extender controllers can safely complete laps on all maps, regardless of friction and difficulty. This is because these maps are well within the distributions that the domain-randomized models

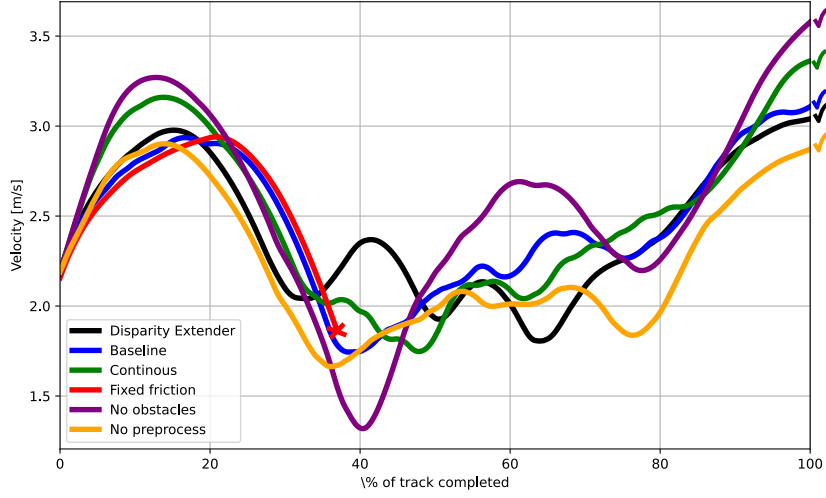


Figure 4.2: Speed profile on Track 3

experienced during training and because the disparity extender was fine-tuned to work well. The fixed friction and no-obstacles model can be seen moving at higher speeds than the successful ones and thus can be observed skidding out of the track, either crashing every lap or completing laps highly unsafely. These models experienced a much more well-observable problem during training than during these tests - they could infer all necessary information from the observation because there was no skidding or suddenly appearing obstacles in their path.

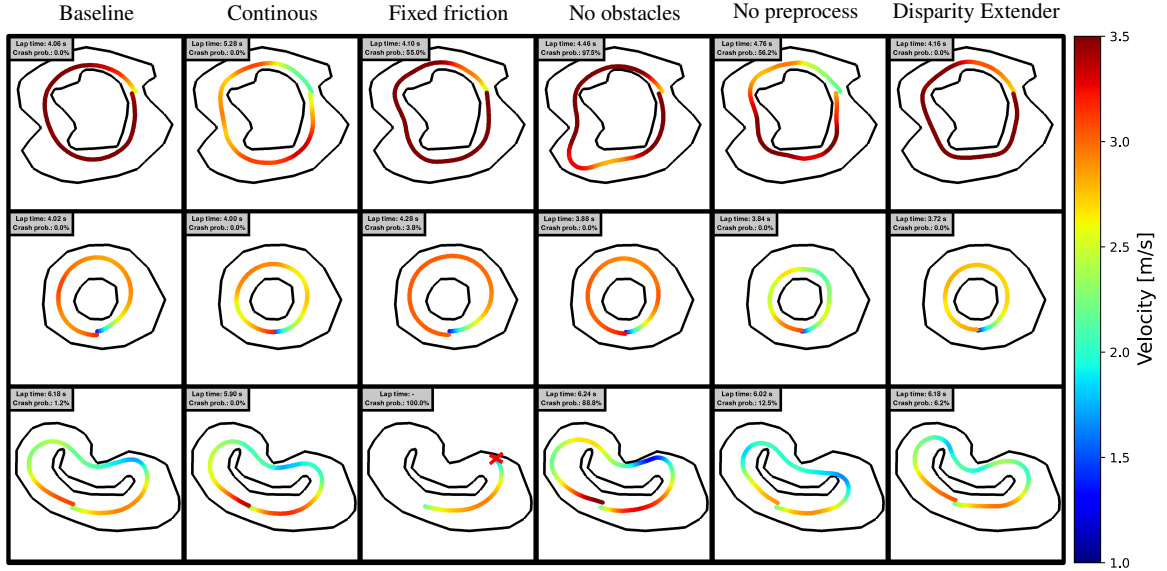


Figure 4.3: The trajectories of the best simulator laps for each model on all the tracks in the counterclockwise direction

4.4 Real-world testing

To measure the performance in the real environment we used two metrics: speed and reliability. We conducted two 10-lap runs for each direction on each of the three tracks. To quantify instability we separated wall touches (touching track edges and moving them by at most 10 cm), and crashes (moving track edges by more than 10 cm) after which we would stop the run and remeasure. If a model crashed five times while completing laps on any single direction of a track, it was categorized as unreliable, though models like this usually couldn't even complete a lap. The 10 lap runs were measured in seconds, and the instability was written as c/w where c denotes the number of crashes, and w is the average wall touches in a lap.

Table 4.1: PERFORMANCE OF THE MODELS IN THE REAL ENVIRONMENTS

Model	Train rew.	Map 1 (5 m/s)			Map 2 (4 m/s)			Map 3 (4 m/s)		
		CW	CCW	Instab.	CW	CCW	Instab.	CW	CCW	Instab.
Baseline (Discrete)	100	4.37	4.55	1/0.3	4.14	4.38	1/0.0	7.26	6.94	2/0.9
Continuous	90	5.66	5.82	3/0.6	3.86	4.50	0/0.0	7.45	7.08	4/1.2
Fixed friction	120	3.85	4.16	0/0.25	-	-	X	8.80	-	X
No dynamic objects	190	-	4.20	X	4.26	4.50	1/0.15	-	-	X
No preprocess	55	-	5.52	X	-	4.61	X	-	-	X
Disparity extender	-	3.93	4.14	0/0.0	3.97	3.89	0/0.0	5.93	6.29	0/0.0

From the results of the experiments (presented in Table 4.1) we can infer the following:

- Without the preprocessing step the policy is both hard to train in the simulator, and unreliable in real environments.
- Training with disturbances (spawning obstacles) in the simulator strengthens the policies' ability to know when to be careful of unknown areas such as sharp corners and rapid turns.
- A model trained with constant friction is stronger than domain randomized models under the grip limit, however becomes dangerously unreliable while slipping.

5 Racing against humans

Reinforcement learning is a biology-inspired method, so comparing its performance against humans seems to be a good benchmark. It is generally said that current racing algorithms can not surpass humans on real vehicles (in simulated situations AI racers can best their human counterparts though as seen in [41]). To test our solution against humans we organized an event, where many people competed against the algorithm. The session took place in a gym, in which we arranged a track as seen in Figure 5.1a. By putting an FPV camera on the car, human drivers could control the car with the help of a Logitech



(a) Track setup



(b) Human driver setup

Figure 5.1: Human vs. AI experiment setup

steering wheel and pedals. By creating a familiar interface (see Figure 5.1b) to work with, inexperienced participants could quickly catch up to more experienced ones.

We did 4 hours of testing against more than ten people, to fairly judge the quality of our agent, relative to humans. We completed time trials and head-to-head races, after which we concluded that our algorithm **is on par with, or slightly better than the average human driver** (which is an impressive feat given that the policy never saw the track during training and the floor was extremely slippery). Drivers who beat our agent found better racing lines, achieving faster laps. Generally, though the RL controller was safer compared to humans. As of lap time, two drivers were able to race faster out of the 20 that tried. The main weakness of our agent was that it could not learn during the test, so it committed the same mistakes again and again (slipping off the optimal racing line, touching walls in certain corners). We are planning to solve this issue by using data-efficient RL algorithms (see more in Section 7), that can learn or adapt while racing.

6 Conclusion

The goal of this work was to train an agent capable of racing effectively in diverse situations on the small-scale autonomous car racing platform F1TENTH, which was achieved. Our model performed better than any learning-based method before it at the International F1TENTH competition in Yokohama, Japan in 2024. Furthermore, various tests and experiments were carried out, comparing and contrasting it to models not using the various training measures we employed (Section 4). Additionally, we conducted a competition against human racers, where we found that our policy is on the level of an above-average driver (without any training on the racetrack).

Our solution consisted of the development of the following elements:

- **Simulator (Section 3.1):** We created a simulator to have full control over the training process, using a simple dynamics model, sensor simulation, with support for RL training using the `stable-baselines3` compatible `gymnasium` API.
- **RL training stack (Section 3.2):** By using a novel preprocessing algorithm we managed to simplify the problem, alleviating the curse of dimensionality. Furthermore, we tested many reward mechanisms, settling on a method circumventing the sparse reward nature of a simple lap time optimization. Additionally, an observation buffer approach was used to increase the agent’s knowledge about a given situation.
- **Bridging the sim2real gap (Section 3.3):** The most difficult problem we faced was the failure of working simulator agents in the real world. We employed several methods during training to improve the sim2real capabilities of our agent, namely domain randomization via perturbation of the environment parameters and learning with disturbances by spawning moving obstacles in front of the car.

Although we are proud of our current racers, we would like to increase their maximal speed, beyond the best non-learning teams. Beyond this, the future directions of this project, include plans to research the possibilities for more sample-efficient training, online learning, and finetuning. Potential directions within and beyond this specific problem are explored in Section 7 in the form of a short position paper. The video of the performance of our car in a different difficult situation can be viewed at: <https://youtu.be/1P7mlawgVDo>

7 Future Work

Although our current method achieved respectable performance, we would like to further improve it. The main goal is to create an agent that continuously learns during its deployment. We found that this objective can be achieved, by either creating a really adaptive agent and training it on multiple domains or developing a hugely sample-efficient RL algorithm that can learn from just a few thousand samples, enabling training to take place entirely in the real world.

7.1 Adaptive RL

Adaptive RL algorithms optimize the evolution capabilities of an agent, making it capable of quickly learning in new environments. One of the main challenges is that of catastrophic forgetting. This occurs when a policy trained on a problem A starts learning on a problem B, and after a while loses its ability to perform on the original problem A. One of the interesting ideas to tackle this is Dynamics-Adaptive Continual Reinforcement Learning via Progressive Contextualization (DaCoRL) [42], an algorithm that uses

a method called Chinese Restaurant Process and clustering in latent dimensions to tell that a new environment has been encountered. It is then categorized into either an environment similar to something previously seen or something entirely new. For every new environment, a new corresponding head is added by expanding the network, initialized to be the same as the head for the nearest similar environment. This means some of the layers are shared, providing for pattern recognition given that the environments are somewhat similar. The researchers find this method to be effective and better performing than the state-of-the-art. Another path is that of Meta Learning, where a policy is trained in multiple environments to be the best possible "learner", being able to quickly discover and master new situations. It is usually assumed, that a set of diverse tasks are available together to train the prior "learner" model on. Online Meta Learning [43] provides a new method that can effectively learn even when tasks are revealed one after another, in the spirit of lifelong learning.

7.2 Sample-efficient RL

The main aim of sample-efficient RL is to learn a policy from tabula-rasa with the least amount of samples. To achieve this goal researchers developed many interesting ideas, but they have a few things in common. Almost all algorithms assume, or try to learn a low-dimensional, high-level representation for the observation, because it is easy to learn and explore in smaller spaces. [44] explored the possibility of using Hierarchical reinforcement learning (HRL) (multiple agents are in a hierarchy, higher level agents give goals to lower level agents, like in a corporate hierarchy). The idea behind using HRL is that high-level policies can learn high-level representations of the problem and explore in the high-level action space, which helps in learning complex tasks with sparse rewards. [45] built on the claim that a good representation should be predictive about future states. With this objective, they defined a self-supervised learning signal for an observation encoder neural network. Testing their method on the Atari 100k benchmark [46] it reached state-of-the-art performance, exceeding expert human performance on several games. [47] attacks the problem of sample-efficient RL from a completely different direction. The authors employ "policy sketches" (sequences of predefined tokens of high-level goals) to learn a policy quickly. The agent in this paper does not know the meaning of the high-level tokens. With this technique, the researchers were able to tackle a small, but complex Minecraft-like game. All in all sample-efficient RL is still an open and interesting problem that we would like to tackle.

8 Acknowledgments

We would like to thank our supervisor Dr. Balázs Németh from HUN-REN SZTAKI, for his huge professional support and hard work helping us in every way possible during our work. We would also like to thank our consultant Dr. Csaba Budai from the BME Department of Mechatronics, Optics and Mechanical Engineering Informatics for his great insights, support, and unending patience during our research.

We are glad to have been supported by the Systems and Control Lab of HUN-REN SZTAKI, the Institute of Computer Science and Control. We especially thank the tiresome work of Professor Péter Gáspár in making our work possible. We are also incredibly grateful for the help we often received from Zoltán Antal and Attila Lelkó.

The research was supported by the European Union within the framework of the National Laboratory for Autonomous Systems (RRF-2.3.1-21-2022-00002).

The research was supported by the EKÖP-24-1 University Research Scholarship Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

References

- [1] M. O’Kelly, H. Zheng, D. Karthik, and R. Mangharam, “F1TENTH: An Open-source Evaluation Environment for Continuous Control and Reinforcement Learning,” in *Proceedings of the NeurIPS 2019 Competition and Demonstration Track* (H. J. Escalante and R. Hadsell, eds.), vol. 123 of *Proceedings of Machine Learning Research*, pp. 77–89, PMLR, 08–14 Dec 2020.
- [2] H. D. Whyte, “Simultaneous localisation and mapping (slam): Part i the essential algorithms,” *Robotics and Automation Magazine*, 2006.
- [3] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. Scokaert, “Constrained model predictive control: Stability and optimality,” *Automatica*, vol. 36, no. 6, pp. 789–814, 2000.
- [4] R. S. Wallace, A. Stentz, C. E. Thorpe, H. P. Moravec, W. Whittaker, T. Kanade, *et al.*, “First results in robot road-following,” in *IJCAI*, vol. 2, pp. 1089–1095, 1985.
- [5] N. Baumann, E. Ghignone, J. Kühne, N. Bastuck, J. Becker, N. Imholz, T. Kränzlin, T. Y. Lim, M. Lötscher, L. Schwarzenbach, *et al.*, “Forzaeth race stack—scaled autonomous head-to-head racing on fully commercial off-the-shelf hardware,” *Journal of Field Robotics*, 2024.
- [6] N. Otterness, “The “disparity extender” algorithm, and f1/tenth,” 2019.
- [7] L. Smith, Y. Cao, and S. Levine, “Grow your limits: Continuous improvement with real-world rl for robotic locomotion,” in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 10829–10836, IEEE, 2024.
- [8] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, *et al.*, “Solving rubik’s cube with a robot hand,” *arXiv preprint arXiv:1910.07113*, 2019.
- [9] T. Haarnoja, B. Moran, G. Lever, S. H. Huang, D. Tirumala, J. Humplik, M. Wulfmeier, S. Tunyasuvunakool, N. Y. Siegel, R. Hafner, *et al.*, “Learning agile soccer skills for a bipedal robot with deep reinforcement learning,” *Science Robotics*, vol. 9, no. 89, p. eadi8022, 2024.
- [10] W. Zhao, J. P. Queralta, and T. Westerlund, “Sim-to-real transfer in deep reinforcement learning for robotics: a survey,” in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, IEEE, Dec. 2020.
- [11] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, “Domain randomization for transferring deep neural networks from simulation to the real world,” in *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pp. 23–30, IEEE, 2017.

- [12] T. Haarnoja, S. Ha, A. Zhou, J. Tan, G. Tucker, and S. Levine, “Learning to walk via deep reinforcement learning,” 2018.
- [13] R. S. Sutton and A. G. Barto, “Reinforcement learning: an introduction, 2nd edn. adaptive computation and machine learning,” 2018.
- [14] C. J. C. H. Watkins, “Learning from delayed rewards,” 1989.
- [15] J.-A. Meyer and S. W. Wilson, *A Possibility for Implementing Curiosity and Boredom in Model-Building Neural Controllers*, pp. 222–227. 1991.
- [16] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, “Curiosity-driven exploration by self-supervised prediction,” *CoRR*, vol. abs/1705.05363, 2017.
- [17] M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, “Unifying count-based exploration and intrinsic motivation,” *Advances in neural information processing systems*, vol. 29, 2016.
- [18] A. W. Moore and C. G. Atkeson, “Prioritized sweeping: Reinforcement learning with less data and less time,” *Machine learning*, vol. 13, pp. 103–130, 1993.
- [19] P. Auer, “Finite-time analysis of the multiarmed bandit problem,” 2002.
- [20] T. Lattimore and C. Szepesvari, “Bandit algorithms,” 2017.
- [21] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [22] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The Bulletin of Mathematical Biophysics*, vol. 5, p. 115–133, Dec. 1943.
- [23] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, “Backpropagation applied to handwritten zip code recognition,” *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.
- [24] J. L. Elman, “Finding structure in time,” *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.
- [25] D. A. Pomerleau, “Alvin: an autonomous land vehicle in a neural network,” in *Proceedings of the 1st International Conference on Neural Information Processing Systems*, NIPS’88, (Cambridge, MA, USA), p. 305–313, MIT Press, 1988.
- [26] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, “Deep reinforcement learning from human preferences,” *Advances in neural information processing systems*, vol. 30, 2017.
- [27] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

- [28] D. P. Kingma, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.
- [29] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” in *ICLR* (Y. Bengio and Y. LeCun, eds.), 2016.
- [30] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017.
- [31] V. Mnih, “Asynchronous methods for deep reinforcement learning,” *arXiv preprint arXiv:1602.01783*, 2016.
- [32] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch, “Emergent tool use from multi-agent autotutorials,” *arXiv preprint arXiv:1909.07528*, 2019.
- [33] X. Sun, M. Zhou, Z. Zhuang, S. Yang, J. Betz, and R. Mangharam, “A benchmark comparison of imitation learning-based control policies for autonomous racing,” in *2023 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1–5, 2023.
- [34] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, “Stable-baselines3: Reliable reinforcement learning implementations,” *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.
- [35] M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. De Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, *et al.*, “Gymnasium: A standard interface for reinforcement learning environments,” *arXiv preprint arXiv:2407.17032*, 2024.
- [36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” *CoRR*, vol. abs/1912.01703, 2019.
- [37] A. Lelkó and B. Németh, “Optimal motion design for autonomous vehicles with learning aided robust control,” *IEEE Transactions on Vehicular Technology*, vol. 73, no. 9, pp. 12638–12651, 2024.
- [38] H. Pacejka and I. Besselink, “Magic formula tyre model with transient properties,” *Vehicle System Dynamics*, vol. 27, no. S1, pp. 234–249, 1997.
- [39] H. Zheng, M. O’Kelly, and A. Sinha, “GitHub - fltenth/fltenth_gym: This is the repository of the F1TENTH Gym environment. — github.com.” https://github.com/fltenth/fltenth_gym.

- [40] T. V. Samak, C. V. Samak, and M. Xie, “Autodrive simulator: A simulator for scaled autonomous vehicle research and education,” in *Proceedings of the 2021 2nd International Conference on Control, Robotics and Intelligent System*, pp. 1–5, 2021.
- [41] F. Fuchs, Y. Song, E. Kaufmann, D. Scaramuzza, and P. Dürri, “Super-human performance in gran turismo sport using deep reinforcement learning,” *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 4257–4264, 2021.
- [42] T. Zhang, Z. Lin, Y. Wang, D. Ye, Q. Fu, W. Yang, X. Wang, B. Liang, B. Yuan, and X. Li, “Dynamics-adaptive continual reinforcement learning via progressive contextualization,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 10, pp. 14588–14602, 2024.
- [43] C. Finn, A. Rajeswaran, S. Kakade, and S. Levine, “Online meta-learning,” in *Proceedings of the 36th International Conference on Machine Learning* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, pp. 1920–1930, PMLR, 09–15 Jun 2019.
- [44] O. Nachum, S. S. Gu, H. Lee, and S. Levine, “Data-efficient hierarchical reinforcement learning,” *Advances in neural information processing systems*, vol. 31, 2018.
- [45] M. Schwarzer, A. Anand, R. Goel, R. D. Hjelm, A. Courville, and P. Bachman, “Data-efficient reinforcement learning with self-predictive representations,” *arXiv preprint arXiv:2007.05929*, 2020.
- [46] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, *et al.*, “Model-based reinforcement learning for atari,” *arXiv preprint arXiv:1903.00374*, 2019.
- [47] J. Andreas, D. Klein, and S. Levine, “Modular multitask reinforcement learning with policy sketches,” in *International conference on machine learning*, pp. 166–175, PMLR, 2017.

Appendix

8.1 Hyperparameters for PPO

The following hyperparameters were changed for the PPO algorithm using the `stable-baselines3` implementation.

- `algo_type`: PPO
- `pi_network_arch`: [64, 64, 64]
- `v_network_arch`: [64, 64, 64]
- `n_steps`: 2^{16} (65 536)
- `n_epochs`: 10
- `batch_size`: 128
- `vf_coef`: 0.4
- `ent_coef`: 0.05
- `gamma`: 0.98
- `gae_lambda`: 0.92

8.2 The constants used for Pacejka’s Magic formula

- $B = 9$
- $C = 1.5$
- $D = \frac{1}{2}m \cdot g \cdot \mu$
- $E = -1$