Informatics

Model-Based Deep Reinforcement Learning for Autonomous Racing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Axel Brunnbauer, BSc.
Matrikelnummer 01452934

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.rer.nat Radu Grosu
Mitwirkung: Dr. Ramin Hasani

Wien, 1. April 2021

Axel Brunnbauer | Radu Grosu

Informatics

Model-Based Deep Reinforcement Learning for Autonomous Racing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering und Internet Computing

by

Axel Brunnbauer, BSc.
Registration Number 01452934

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.rer.nat Radu Grosu
Assistance: Dr. Ramin Hasani

Vienna, 1st April, 2021

Axel Brunnbauer	Radu Grosu

Erklärung zur Verfassung der Arbeit

Axel Brunnbauer, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. April 2021

Axel Brunnbauer

Danksagung

An dieser Stelle möchte ich mich zuerst bei meinen Betreuern Radu Grosu sowie Ramin Hasani für deren Unterstützung, Geduld sowie die anregenden Diskussionen bedanken.

Ganz besonderer Dank gilt vor allem auch meinen Kollegen Luigi Berducci und Andreas Brandstätter, ohne deren Unterstützung wäre diese Arbeit nicht möglich gewesen.

Schließlich möchte ich mich auch bei meiner Familie sowie meinen Freunden für deren Unterstützung bedanken.

Acknowledgements

At this point, I would first like to thank my supervisors Radu Grosu and Ramin Hasani for their support, patience and interesting discussions.

Special thanks also go to my colleagues Luigi Berducci and Andreas Brandstätter, without their support this work would not have been possible.

Finally, I would like to thank my family as well as my friends for their support.

Kurzfassung

Reinforcement Learning (RL) ist zurzeit eines der aktivsten Forschungsfelder im Bereich des maschinellen Lernens und hat bereits in zahlreiche Anwendungsgebiete, wie etwa in die Steuerung von autonomen Fahrzeugen oder in die intelligente Produktempfehlung, Einzug gehalten. Trotz der vielen Fortschritte der vergangenen Jahre, gibt es noch einige Hürden die es zu überwinden gilt, bevor RL in industriellen Anwendungen eingesetzt werden kann. Zu diesen Hürden zählt etwa die schier endlose Menge an Trainingsdaten, die benötigt wird um Agenten (Teilnehmer eines Entscheidungsprozesses) mithilfe von RL zu trainieren. Eine bestimmte Art von RL Algorithmen beschäftigt sich mit der Idee, ein Modell anhand der Daten zu lernen, um damit neue Trainingsdaten zu generieren. Diese Art von RL wird auch modellbasiertes RL genannt und man verspricht sich davon die Menge der benötigten Trainingsdaten auf ein Maß zu reduzieren, das es erlaubt RL Algorithmen in Umgebungen zu trainieren, in denen es schwierig ist, ausreichend Daten zu generieren.
Das Ziel dieser Arbeit ist es, die Vorteile, die modellbasierte RL Algorithmen mit sich bringen, zu untersuchen. Dazu adaptieren wir im Folgenden einen existierenden, modellbasierten RL Algorithmus und vergleichen dessen Performance mit der von gängigen, nicht modellbasierten RL Algorithmen, die den derzeitigen State-of-the-Art markieren. Die Anwendungsdomäne in der wir die Experimente durchführen ist im Bereich des autonomen Fahrens angesiedelt. Um genauer zu sein werden die Agenten darauf trainiert, Rundenzeiten in Zeitrennen zu minimieren. Die Experimente zielen darauf ab, Algorithmen, die in der Simulation trainiert wurden, dahingehend zu beurteilen, wie gut ihre Fähigkeit ist, auf unvorhergesehene Ereignisse in der echten Welt zu reagieren. Außerdem vergleichen wir die Flexibilität der Algorithmen, auf anderen, ungesehenen Strecken vergleichbare Ergebnisse zu liefern. Nicht zuletzt untersuchen wir auch das Trainingsverhalten der unterschiedlichen Algorithmen. Die Experimente werden sowohl in einer eigens für diese Arbeit implementierten Simulationsumgebung, als auch auf einer Prototyping Plattform, die auf einem kleinen, ferngesteuerten Auto basiert, durchgeführt.

Abstract

Reinforcement learning (RL) is currently one of the most active machine learning research fields. RL algorithms have been successfully deployed ubiquitously in many real-world application domains, such as autonomous vehicles, intelligent production sites, and finance. Despite the many advances made in recent years, there are still fundamental challenges that need to be addressed before RL can be reliably applied in industrial applications. One of these problems is the sheer amount of training data that is needed to train deep RL agents. Model-based RL is a branch of RL algorithms that learn a model of the agent or its environment which is then leveraged to generate new training data or to plan ahead. Model-based approaches are expected to reduce the required amount of training data to be sampled from an environment, down to a level that allows RL algorithms to be trained in environments where it is hard to generate sufficient data. The goal of this work is to investigate the advantages that model-based RL algorithms bring. To this end, we adapt an existing model-based RL algorithm and compare its performance with that of common, model-free RL algorithms that mark the current State-of-the-Art. The application domain in which we conduct the experiments is in the field of autonomous racing. In our experiments, agents are trained to minimize lap times in time-trial races. The experiments aim to evaluate algorithms, that were trained in simulation, with respect to their ability to be deployed in the real world. We also compare the flexibility of the algorithms to produce comparable results on other, unseen race tracks. Last but not least, we also investigate the training behavior of the different algorithms. The experiments are performed both in a simulation environment, implemented specifically for this work, and on a prototyping platform based on a small remote-controlled car.

Contents

CHAPTER 1

Introduction

In the following sections, we introduce the topic of this work and motivate the problem we address throughout the next chapters. First, in section 1.1, we set forth the importance and relevance of this work and the area of research where this work is located in general. Then, we formulate the problem and the questions we try to address and answer in section 1.2. Finally, in section 1.3, we give a brief outline of how this work is structured and how the content is organized.

1.1 Motivation

Building intelligent autonomous agents has been a major goal for computer science researchers for several decades now. One important aspect of intelligent agents is their capability of clever decision-making when interacting with their environment. Reinforcement Learning (RL) is a branch of Machine Learning (ML) research, that addresses decision-making strategies of such agents. Although being around for several decades, RL research began to make significant progress just a few years ago by courtesy of advances in other fields of ML, such as supervised or unsupervised learning. Now, the field of RL was among the most active areas of research in recent years. Research projects such as AlphaGo [SHM$^+$16] or AlphaStar [VBC$^+$19] gained a lot of media coverage when they proved superhuman capabilities in games like Go and StarCraft and beat the world's elite. This success initiated an outstanding increase in publications and contributions in the field of RL.

While the state of the art advances from year to year, significant, unsolved problems remain. Its inherent data inefficiency and hard to reproduce results prevent RL to be widely adopted by the industry yet. Furthermore, it is not yet clear how to teach agents or even formulate which situations are undesirable without ever experiencing them. This is especially true for the field of robotics and autonomous driving, where it is not feasible to let robots or vehicles crash many million times until the agent learns to prevent such situations. A common approach is to train agents in simulated environments before deploying them to real-world platforms. However, even when high-fidelity simulations are available, which is difficult to achieve in highly unpredictable domains (e.g urban traffic), RL algorithms often face difficult challenges due to their susceptibility to distributional shift between simulation and reality.

1.2 Problem Statement

In this work, we aim to evaluate algorithms of two different branches in RL with respect to their capability to generalize behavior in simulation and then successfully transfer this knowledge to real-world vehicles. In the course of experimental evaluations, we compare state-of-the-art model-free RL algorithms with a recent, advanced model-based RL algorithm in the context of autonomous racing. While model-free approaches tend to outperform current model-based approaches in the limit, the latter ones might be better suited to tackle unsolved challenges, such as high sample complexity and sensitivity to distributional shift. To this end, we set out the following research questions, which we try to answer throughout this work:

1. How do world-model RL algorithms compare to the state-of-the-art model-free RL algorithms concerning sample efficiency and performance?
2. Are world models able to adapt to unknown situations at test time?
3. Do world models, that are learned in simulation, facilitate safe real-world navigation for autonomous vehicles?

To answer these questions, we conduct a series of experiments in the domain of autonomous racing, both in simulation and on our real-world miniature race car, which serves as a testbed for autonomous driving scenarios. The goal is to develop a system, that is capable of learning to navigate difficult racetracks just from observations in an end-to-end manner. Our approach is based on Dreamer [HLBN20], an existing model-based RL algorithm, which we adapted to the domain of autonomous racing. We compare our approach to various model-free RL algorithms for continuous control RL. We compare these algorithms to our approach with respect to their sample efficiency, generalization capabilities, and performance. To summarize, the main contributions of this work are:

- We demonstrate the effectiveness of advanced model-based deep RL compared to model-free agents in the real-world application of autonomous racing.
- We show the transferability of advanced model-based deep RL agents to real-world applications where model-free agents fail.
- We empirically show that the learning performance and generalization ability of Dreamer depend on the choice of the observation model and its ability to learn a meaningful dynamics model.

1.3 Outline

In chapter 2, we give a concise introduction to the fundamentals of Markov Decision Processes (MDPs) and RL in general. We cover basic concepts and algorithms underlying most of the more advanced approaches before moving on to introducing state-of-the-art model-free algorithms, which serve as a performance baseline in our experiments.
Subsequently, in chapter 3, we discuss related work in RL for robot control in general and the state of the art in autonomous racing. Furthermore, we present two existing approaches of end-to-end RL in this domain.
Chapter 4 introduces the model-based algorithm we adapted to the domain of racing before discussing the algorithmic contributions we made. These contributions are two novel reconstruction models to train the algorithm and the formal description of the reward signal we used during the training of the agents.
In chapter 5, we outline the infrastructure we built to train agents and conduct experiments. Specifically, we present the simulation environment we designed, the hardware platform used to evaluate the real-world performance and the race tracks on which we train and evaluate agents. Furthermore, we outline the training and hyperparameter tuning procedure, as well as the algorithm implementations and model architectures.
The experimental design and results are presented and discussed in chapter 6.
Chapter 7 concludes the contributions and results obtained throughout this thesis and gives an outlook on our current and future research work in this field.

CHAPTER 2
Background

The following sections provide an overview of important key concepts that are fundamental to most of the approaches in the field of RL and robot learning. First, we introduce MDPs, a powerful framework to model different kinds of sequential decision-making problems. Subsequently, we provide insight into the field of RL, a branch of ML concerned with finding optimal behaviors in decision-making problems, formulated as MDPs. The last part of this chapter introduces and motivates the choice of the RL algorithms we use in our experiments.

2.1 Markov Decision Processes

Modeling dynamical systems under uncertainty requires the availability of a stochastic model which is capable of taking random events into account. The field of probability theory provides a variety of stochastic models which can be used to predict the behavior of dynamical systems under uncertainty. To formally define sequential decision-making processes, a common approach is to formulate the underlying system as a MDP. Intuitively, MDPs model the interaction of an agent with its environment. There exist different types of MDPs, such as continuous-time, infinite, or constrained variants. In this work we focus on discrete time MDPs and Partially Observable MDPs (POMDPs).
Figure 2.1 shows the schematics of MDPs: agents observe their environment and obtain the current state s_t as well as a reward r_t at some discrete time step t. Rewards can be understood as an incentive to reach favorable states while avoiding undesirable ones. Agents then decide which action a_t they should take in order to maximize their rewards. After executing the action, the environment is potentially altered and agents observe a new state and reward.

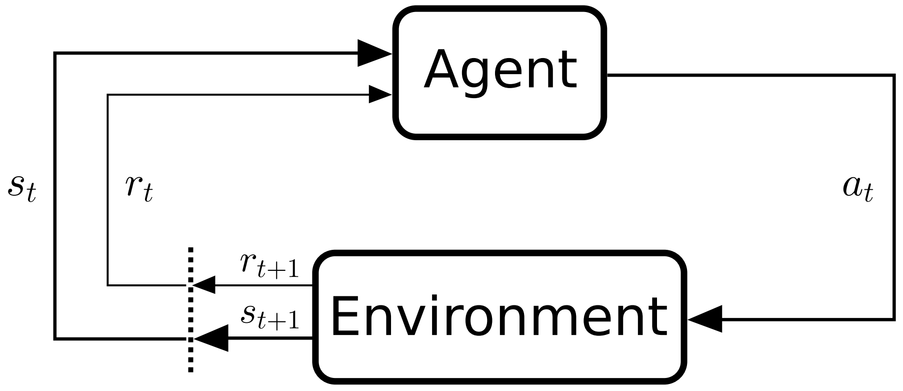


Figure 2.1: In a Markov Decision Processes, agents perceive states s_t and rewards r_t at time step t, and interact with their environment via actions a_t. [SB18]

2.1.1 Fully observable MDPs

MDPs can be formally defined as a 4-tuple $\langle S, A, T, R\rangle$. The set S contains all states that are reachable for an agent. The set of actions that an agent can execute is defined by A. States and actions can be either discrete or continuous and can be of arbitrary dimension. The dynamics model of the system is defined by the transition function T which is defined as

$$T\colon S \times A \times S \mapsto [0,1]. \tag{2.1}$$

Thus, T is a stochastic function that expresses the probability of reaching some state s_{t+1}, given the agent is in state s_t and executes action a_t. This relationship is usually expressed as a conditional probability distribution:

$$T(s_t, a_t, s_{t+1}) = p(s_{t+1}|s_t, a_t). \tag{2.2}$$

It is valid to assume that the next state s_{t+1} only depends on the previous state s_t and action a_t, because, like any Markov model, MDPs are assumed to satisfy the Markov property. To satisfy the Markov property, the state of a system must be conditionally independent from any other previous states except its immediate predecessor. Formally, this can be written as

$$p(s_{t+1}|s_{0:t}, a_{0:t}) = p(s_{t+1}|s_t, a_t). \tag{2.3}$$

Rewards are generated by a deterministic function R, which assigns a scalar reward value to an action a_t taken in some state s_t. Thus, R is defined as

$$R\colon S \times A \mapsto \mathbb{R}. \tag{2.4}$$

In this work, we defined the dynamics to be stochastic, while rewards are generated by a deterministic function. Other works might define rewards to be also stochastic or the dynamics to be deterministic. Sometimes, R also depends on the next state that follows after taking action a_t. These assumptions are problem-dependent and in the following, we will use the definitions above, although other interpretations might be also valid.

2.1.2 Partially Observable MDPs

Given the definitions above, many scenarios can be described as MDPs. In real-world scenarios, however, agents are not able to observe the true state of a system directly. Observations that are obtained by sensors, such as RGB cameras or Light Detection and Ranging (LiDAR) systems, are subject to noise. Moreover, such measurements might not be able to capture the full state of a system, and thus only *partially* observe the true state. Figure 2.2 shows an example of a vehicle that obtains a LiDAR measurement of its environment. In this visualization, it is clear that other vehicles probably occlude other traffic participants. Furthermore, one can not deduct important components of the vehicle state (e.g. pose, velocity) from a single observation. Thus, the state of the environment is partially observable and the problem is formulated as a POMDP.

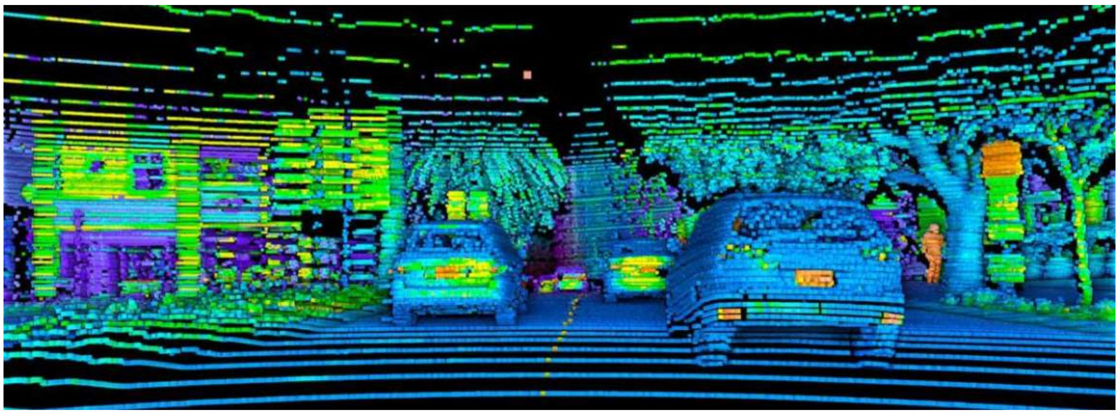


Figure 2.2: Traffic scene captured by a LiDAR sensor.[1]

POMDPs can be defined as a 6-tuple $\langle S, A, T, R, \Omega, O\rangle$ [Ås65, KLC98]. The entities S, A, T and R describe the underlying MDP. Additionally, POMDPs define the set of observations Ω. The function O, which is defined as

$$O\colon \Omega \times S \times A \mapsto [0,1], \tag{2.5}$$

maps an observation together with a state-action pair to a probability. Often, this relationship is expressed as

$$O(o_t, s_t, a_t) = p(o_t | s_t, a_t). \tag{2.6}$$

[1]Source: automotiveworld.com, accessed 2021-03-27

Intuitively, this expresses the likelihood of obtaining an observation o_t in the true state s_t while executing action a_t.
Often it is convenient to treat POMDPs as MDPs with *belief states*. Because states can not be observed directly, a belief over the true state is estimated from observations. In most cases, this belief is represented as a probability distribution. Maintaining this belief over time is usually achieved by applying recursive Bayes filters:

$$\overline{\text{bel}}(s_{t+1}) = \eta \, p(o_t|s_t, a_t) \int_{s_t} p(s_{t+1}|s_t, a_t) \, \overline{\text{bel}}(s_t) \, ds_t. \tag{2.7}$$

In eq. (2.7), the distributions correspond to the transition (eq. (2.2)) and observation model (eq. (2.6)). The constant η denotes a normalization factor. There exist various filtering methods, such as Kalman or Particle Filters, which are tractable approximations to the recursive Bayes filter formulation in eq. (2.7) [TBF05].

2.2 Introduction to Reinforcement Learning

Continuing on the concepts introduced in the previous section, we now focus on methods that enable agents to learn desired behavior in any environment. First, we introduce the fundamentals of RL. Then, we introduce important RL algorithms which we used as baselines to test our approach on a racing task. For this section, most of the definitions and notation are based on the work presented in [SB18, Ser21, Ope21].

2.2.1 Fundamentals

When interacting with their environment, agents produce a trajectory of states and actions. Such trajectories, often denoted as τ, are of the form $s_0, a_0, s_1, ..., s_T$. The goal of RL is, to find a behavior generating function, a policy, that maximizes the collected rewards over time. Formally, an agent aims to maximize the sum of rewards, denoted by $G(\tau)$:

$$G(\tau) = \sum_{t=0}^{T} R(s_t, a_t) \tag{2.8}$$

In eq. (2.8), the sum is taken over a horizon T, starting from time step $t = 0$. MDPs can be considered over a finite horizon (episodic MDPs) or infinite horizon. For episodic tasks, T is finite, while for infinite time-horizon tasks $T = \infty$. For infinite time-horizon tasks, rewards are discounted by a factor $\gamma^t \in [0, 1]$, that weighs the relative importance of future rewards by transforming $G(\tau)$ in a geometric series that converges in the limit. The rationale behind this is that rewards, that are to be collected far in the future, should not influence current decisions as much as immediate rewards.

Policies. The function that produces actions is usually referred to as *policy*. Policies are functions that map states to actions and can be either deterministic or stochastic. In this work, we usually assume stochastic policies, as they are more general than deterministic ones. Thus, a policy is defined as

$$\pi\colon A \times S \mapsto [0, 1]. \tag{2.9}$$

Policies can be interpreted as mappings from states s_t to the probability of taking action a_t in time step t. This is usually expressed as a conditional probability distribution and often denoted as $\pi(a_t|s_t)$.

Value functions, Q-values, Advantages. To estimate how desirable it is to reach a state s_t, *value functions* are computed. A value function, usually denoted as V^π, gives an estimate of the expected sum of returns when starting from some state s while following a policy π. Formally, this can be written as:

$$V^\pi(s) = \mathbb{E}_{\tau\sim p(\cdot|\pi)}\big[\ G(\tau)\ \big|\ s_0 = s\ \big] \tag{2.10}$$

$$p(\tau|\pi) = p(s_0, a_0, ..., s_T|\pi) = p(s_0) \prod_{t=0}^{T} \pi(a_t|s_t)\ p(s_{t+1}|s_t, a_t) \tag{2.11}$$

In eq. (2.10) we compute the expected sum of returns considering actions and states following the distribution over trajectories. The trajectory distribution is defined in eq. (2.11) and is governed by the initial state distribution $p(s_0)$, the policy π and the transition model p, which we defined in eq. (2.2).
Analogous to value functions, we can compute state-action values, also often referred to as Q-values:

$$Q^\pi(s, a) = R(s, a) + \mathbb{E}_{s'\sim p(\cdot|s,a)}\big[V^\pi(s')\big]. \tag{2.12}$$

Q^π computes the expected sum of returns when taking action a in state s and then following a policy π. By using Q-values and value functions, we can also compute the advantage of taking an action a in state s instead of sampling an action from the policy π. Formally, the advantage is defined as

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s). \tag{2.13}$$

In MDPs with small, discrete state and action spaces, policies and value functions can be computed with dynamic programming algorithms [Bel03]. These approaches are also known as policy iteration algorithms. However, when dealing with high-dimensional or continuous state and action spaces, tabular methods are not feasible and value functions and policies have to be approximated [SB18, Ber07]. Current methods usually make use of deep neural networks as general function approximators, hence they are often referred to as deep RL algorithms. When using deep neural networks as policies or value functions, we denote the parameters of the networks as θ or ψ, respectively. The notation for policies or value functions is then π_θ or V_ψ, respectively.

On-policy vs. Off-policy. In RL, one distinguishes between on-policy and off-policy algorithms. On-policy algorithms perform updates on the same policy that is used to interact with the environment. Off-policy algorithms can update a policy and estimate value functions that are different from the policy that is used to generate sample data. Thus, off-policy approaches are usually more sample efficient, as they can reuse data sampled from older versions of the policy. Also, such algorithms can leverage data that is produced by a modified policy. For instance, it is common to use more random policies for data collection to improve the exploration behavior of agents. However, these advantages often come at the price of algorithms that are less likely to converge [SB18].

RL Algorithms Overview. In recent years, a diverse landscape of RL algorithms emerged. Figure 2.3 gives a brief, but incomplete overview of current RL algorithms. Generally, RL algorithms are classified as model-free or model-based. Model-free algorithms do not use any model of their environment, such as the transition model defined in eq. (2.2). In contrast, model-based algorithms have such a model available, either given or learned from data. Having a model of the decision process makes it possible to reason about the consequences of actions more directly. However, the fidelity of predictions depends heavily on the quality of such a model.

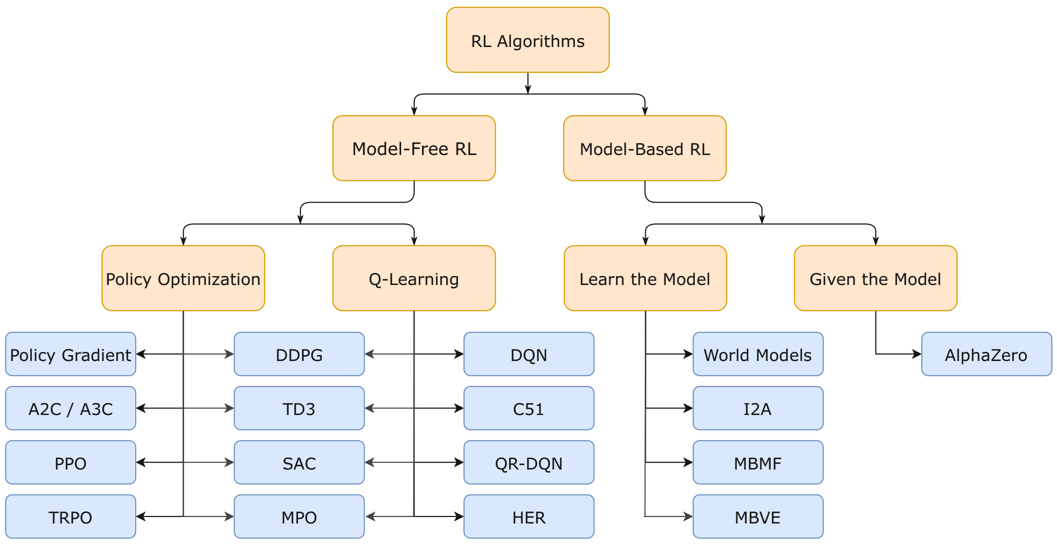


Figure 2.3: Landscape of RL algorithms. [2]

Model-free RL algorithms can be further distinguished by the way they learn: *Policy Optimization* algorithms directly optimize parameterized policies, *Q-Learning* refers

[2]source: spinningup.openai.com, accessed 2021-03-30, modified

to algorithms that learn value- or Q-functions to derive policies from them. Often, approaches try to combine policy optimization with Q-Learning. Such methods are commonly known as *Actor-Critic* methods. However, most algorithms can not be clearly assigned to one category. Therefore, fig. 2.3 is just a coarse categorization and should provide only a rough overview.
From the algorithms shown in fig. 2.3, PPO [SWD+17], DDPG [BMHB+18, LHP+15], SAC [HZAL18] and MPO [AST+18] are of particular interest, as they serve as performance baselines to our approach, which is a world-model algorithm.

2.2.2 Basic Algorithms

To understand the algorithms referenced in this work, we briefly present the underlying concepts behind most advanced RL algorithms. In their cleanest form, two types of RL algorithms are distinguished: policy gradient methods and Q-learning approaches. Generally, all approaches aim to solve the same optimization problem:

$$\pi^* = \arg\max_{\pi} V^{\pi}(s), \quad \forall s \in S. \tag{2.14}$$

Q-Learning. In its simplest form, Q-learning is a simple dynamic programming algorithm and designed for environments with discrete, finite and small state and action spaces.

Algorithm 2.1: Q-Learning [SB18]

```
1  Input: step size $\alpha \in (0,1]$, exploration factor $\epsilon \in [0,1]$, discount factor $\gamma \in (0,1]$
2  $\forall s \in S, \forall a \in A$: set $Q(s,a)$ to arbitrary value.
3  $\forall s \in S_{\text{terminal}} : Q(s,\cdot) = 0$.
4  while not converged do
5      Sample random starting state $s_0$.
6      Set $t = 0$.
7      while $s_t$ is not terminal do
8          Select $a_t$ from behavior policy $\pi$ (e.g. $\epsilon$-greedy).
9          Observe $s_{t+1}, r_t$ by taking action $a_t$ in state $s_t$.
10         $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\Big(r_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\Big)$
11         $t \leftarrow t+1$.
12     end
13 end
```

Q-Learning is an off-policy algorithm, thus it can estimate Q-values for any policy π. In algorithm 2.1, line 10 shows the update rule for the Q-values. Each update corrects the estimation of the Q-values proportional to the quantity

$$r_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t), \tag{2.15}$$

which is also known as *Bellman error*. For tabular state and action spaces and under the assumption that every state has a non-zero probability to be visited under π, Q-Learning is guaranteed to converge, thus eq. (2.15) approaches zero.

Vanilla Policy Gradient. In contrast to action-value methods, such as Q-learning, policy optimization methods do not rely on learning state or state-action value functions. Instead, they directly optimize a parameterized policy with respect to their parameters, using any optimization algorithm. In the following, we will focus on *policy gradient* algorithms, which use *gradient ascent* as a procedure to solve the optimization problem defined in eq. (2.14). Other works also make use of gradient-free optimization techniques [SHC+17].
The gradient ascent update for the policy parameters θ is of the form

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta), \tag{2.16}$$

where θ are the policy parameters, α the learning rate and $J(\theta)$ is the objective function, which is usually some variant of the expected sum of returns, as defined in eq. (2.10):

$$J(\theta) = \mathbb{E}_{\tau\sim p(\cdot|\pi_\theta)}[G(\tau)] \tag{2.17}$$

Taking the gradient of eq. (2.17) with respect to θ yields:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p(\cdot|\pi_\theta)}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\ G(\tau)\right] \tag{2.18}$$

There exist improvements for the objective defined in eq. (2.17), dealing with the typically high variance of this formulation. A simple approach is to consider only future rewards for computing $G(\tau)$. Another approach is to subtract a state-dependent baseline value (such as $V(s_t)$) from $G(\tau)$. This can reduce the gradient variance significantly, but also introduces some bias. The latter approach is also known as *actor-critic* method, where the policy is the actor and the value function is referred to as critic, as it judges the actions taken by the actor. Many policy gradient algorithms use the advantage function A^π as an objective, which naturally uses the baseline subtraction, shown in eq. (2.13). Thus, the corresponding policy gradient has the following form:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p(\cdot|\pi_\theta)}\left[\sum_{t=0}^{T} \Big(\nabla_\theta \log \pi_\theta(a_t|s_t)\Big)\ A^{\pi_\theta}(s_t, a_t)\right]. \tag{2.19}$$

Algorithm 2.2 shows the basic policy gradient algorithm, using a state-dependent value baseline with advantage estimation. For each update step k, a set of trajectories $\mathcal{D}_k$ is sampled from the environment by executing policy π_{θ_k}. Then, the advantages of actions in each time step are estimated from the learned value function and collected data. Next, in line 5, the estimation of the policy gradient by averaging the gradients over the set of trajectories is depicted. Having computed the gradients, the policy parameters θ are

Algorithm 2.2: Vanilla Policy Gradient Algorithm [Ope21]

```
1 Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$.
2 for $k = 0, 1, 2, ...$ do
3     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_{\theta_k}$ in the environment.
4     Compute advantages $A_t$ from $V_{\phi_k}$ (using GAE [SML+16] or other methods).
5     Estimate policy gradient using Monte-Carlo approximation (see eq. (2.19)):
          $$\nabla_{\theta_k} J(\theta_k) \approx \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_{\theta_k} \log \pi_{\theta_k}(a_t|s_t) A_t.$$
6     Update policy weights using stochastic gradient ascent (or other optimizers,
        such as Adam [KB15]):
          $$\theta_{k+1} = \theta_k + \alpha \nabla_{\theta_k} J(\theta_k).$$
7     Fit value function by regression on sum of future rewards:
          $$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - G(\tau_{t:T})\right)^2.$$
8 end
```

updated, as shown in line 6. In the last step, the value function is fitted to the sum of future rewards using the Mean Squared Error (MSE) loss over trajectories. Note that in line 7, the sum of returns is computed only for $\tau_{t:T}$, which is the trajectory starting from time step t until the end of the trajectory at time step T. The term $G(\tau_{t:T})$ is also known as *rewards to go*, as it computes the sum of rewards from the current time step.

2.3 Model-Free Baseline Algorithms

Based on the vanilla policy gradient and Q-learning algorithms, many more advanced RL algorithms emerged in recent years and successively improved the state of the art on common benchmarks. We chose four well established model-free RL algorithms to provide a competitive baseline in end-to-end learned racing performance. In the following, we briefly introduce the algorithms and their main advantages over other algorithms. In addition to the corresponding papers, we often make use of the notation and excellent explanations provided in [Ope21].

2.3.1 Proximal Policy Optimization

Vanilla policy gradient algorithms use gradient ascent, a first-order optimization procedure, to update the parameters of policies. However, when using highly non-linear functions, first-order approximations are often too inaccurate, leading to drastic performance collapses when stepping too far from the current policy during updating. Thus, approaches that use concepts from mathematical optimization, such as trust-region optimization, emerged. A prominent example of such approaches is Trust-Region Policy Optimization (TRPO), an algorithm that takes the largest possible update step within a trust-region [SLA+15]. The trust-region is defined by the Kullback-Leibler (KL) divergence between the current and the updated policy. However, the computation of the trust-region can be computationally demanding, as it involves the computation of Hessian matrices [EIS+20]. Schulman et al. proposed a new family of policy optimization algorithms, Proximal Policy Optimization (PPO), that is able to constrain the size of the update step efficently [SWD+17]. Furthermore, PPO is simpler than TRPO and thus implementations are less error prone.

PPO modifies the objective of the vanilla policy gradient algorithm, depicted in algorithm 2.2:

$$J(\theta) = \mathbb{E}_{\tau \sim p(\cdot|\pi_{\theta_k})}\left[\sum_{t=0}^{T} \min\Big(r_t(\theta)A_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\Big)\right], \tag{2.20}$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, \tag{2.21}$$

$$A_t = A^{\pi_{\theta_k}}(s_t, a_t). \tag{2.22}$$

In eq. (2.20), π_{θ_k} denotes the current, fixed policy (after k update steps), while θ are the current parameters to be evaluated. The quantity $r_t(\theta)$ denotes the probability ratio of taking action a_t in state s_t under the policies π_θ and π_{θ_k}. For instance, if it was more likely to take action a_t in state s_t under policy π_{θ_k} than under the new policy π_θ, then $r_t(\theta) < 1$. If A_t is positive, thus it is favorable to take action a_t, then PPO encourages $r_t > 1$, but only up to a value of $1 + e$ to avoid updates that are too large. Similarly, PPO avoids updates that deviate too much from the old policy when A_t is negative.

The reason we chose this baseline algorithm is its simplicity and good performance on continuous control tasks. Furthermore, we also aimed to have at least one on-policy algorithm in our collection of benchmark algorithms. We are aware that there exist newer on-policy algorithms, such as V-MPO [SAS+20], but we decided to use an existing, well-tested implementation of PPO over self-implemented or proof-of-concept implementations.

2.3.2 Deep Deterministic Policy Gradients

In contrast to PPO, Deep Deterministic Policy Gradient (DDPG) learns a deterministic policy μ. Furthermore, DDPG is a Q-Learning-based, off-policy algorithm. Similar to the

DQN-Algorithm (Deep Q-Network; [MKS$^+$15]), DDPG approximates the Q-function by a deep neural network, but for continuous action spaces. The learned Q-function Q_ϕ is then used to update the policy parameters θ by backpropagation through Q_ϕ. Formally, the corresponding gradient of the objective function for the policy parameters is given by

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p(\cdot|\mu_\theta)}\left[\sum_{t=0}^{T} \nabla_\theta\, Q_\phi(s_t, \mu_\theta(s_t))\right]. \tag{2.23}$$

In addition to the objective function depicted in eq. (2.23), DDPG applies several, well established tricks, such as target networks and replay buffers [LHP$^+$15].
In this work we use Distributed Distributional DDPG (D4PG), an algorithm which is a *distributional* extension of DDPG [BMHB$^+$18]. Instead of a deterministic Q-function, D4PG learns the parameters of a distribution for a random variable Z for the Q-values. The state action value is the expectation over Z:

$$Q_\pi(s,a) = \mathbb{E}[Z_\pi(s,a)] \tag{2.24}$$

Additionally, D4PG also allows multiple, distributed actors to run in parallel to collect experience.

We chose D4PG as a baseline algorithm because it demonstrated good performance on a variety of simulated control tasks, such as the *Deepmind Control Suite*, and also served as a baseline algorithm for Dreamer [HLBN20].

2.3.3 Soft Actor-Critic

Soft Actor-Critic (SAC) is an off-policy actor-critic algorithm that makes use of stochastic policy optimization but also incorporates core ideas from DDPG algorithms. Like the Twin-Delayed DDPG algorithm [FvHM18], another extension of DDPG, SAC learns two Q-functions, which are used to computed the optimization objective. SAC is a maximum entropy RL algorithm that adds an entropy regularization term to the objective, incentivizing as much explorative behavior while still maximizing rewards.
The implementation of SAC we use in this work learns two Q-functions concurrently, instead of a Q-function and a value function [HZAL18, Ope21]. SAC slightly modifies the Q-function definition by adding an entropy term to the step rewards:

$$Q^\pi(s,a) = \mathbb{E}_{\tau\sim p(\cdot|\pi)}\left[\sum_{t=0}^{T} R(s_t,a_t) + \alpha \sum_{t=1}^{T} H(\pi(\cdot|s_t))\,\middle|\, s_0 = s, a_0 = a\right]. \tag{2.25}$$

SAC learns a stochastic policy π_θ. To sample from π_θ, SAC learns two deterministic functions for the mean and standard deviation parameters and applies the reparameterization trick. Thus, sampling is done by scaling the standard deviation by random noise and adding it to the mean:

$$a_\theta(s,\xi) = \tanh(\mu_\theta(s) + \sigma_\theta(s) \odot \xi). \tag{2.26}$$

In eq. (2.26), the sampled value is squashed by a hyperbolic tangens to constrain the action values to the interval $[-1, 1]$. The resulting objective function for the policy is defined by

$$J(\theta) = \mathop{\mathbb{E}}_{s\sim D,\, \xi\sim\mathcal{N}(0,I)} \left[\min_{j=1,2} Q_{\phi_j}(s, \tilde{a}_\theta(s,\xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,\xi)|s) \right]. \tag{2.27}$$

In eq. (2.27), states are sampled from some collection of previous transitions (e.g. replay buffers) and ξ is sampled from a standard normal distribution. The objective is, similar to DDPG, to choose actions that maximize the Q-values. Note that we take the minimum of the evaluations of the two learned Q-functions. This is often referred to as *Clipped Double-Q Learning.* [FvHM18]. The next term is the entropy regularization term. Note that instead of adding the entropy, we now subtract the log-likelihood of taking the sampled action in state s. This is simply a reformulation which makes use of the definition of entropy: $H(P) = \mathbb{E}_{x\sim P}[-\log P(x)]$.

Our decision to include SAC as a baseline was based on the insensitivity with respect to hyperparameters. Moreover, SAC was successfully deployed and trained on real-world robots [HZH+19].

2.3.4 Maximum a Posteriori Policy Optimization

Maximum a Posteriori Policy Optimization (MPO) is an off-policy RL algorithm that puts the problem of finding optimal actions in the context of probabilistic inference and was proposed in [AST+18]. The idea is to formulate the problem as an Expectation Maximization (EM) problem, for which powerful approaches from probabilistic estimation are known. Intuitively, instead of asking which actions should be taken in order to maximize future rewards, this approach asks which actions were most likely taken, assuming optimality in the future.
The corresponding likelihood function, depicted in eq. (2.28), models how likely it is to act optimally when taking specific actions and is assumed to be proportional to the sum of rewards. The parameter α is a temperature, scaling the individual contributions of each reward.

$$p(\mathcal{O} = 1|\tau) \propto \exp \sum_{t=0}^{T} \frac{R(s_t, a_t)}{\alpha} \tag{2.28}$$

$$p_\pi(\mathcal{O} = 1) = \log \int_\tau p(\tau|\pi)\, p(\mathcal{O} = 1|\tau)\, d\tau \tag{2.29}$$

By applying the rule of marginal probability, we can obtain the likelihood of optimality, when following a policy π, shown in eq. (2.29). Since the integral is not tractable, it is approximated by the evidence lower bound (ELBO). The resulting lower bound J, which is to be maximized, is given by:

$$J(q,\theta) = \mathbb{E}_{\tau\sim p(\cdot|q)} \left[\sum_{t=0}^{\infty} \gamma^t \Big(R(s_t, a_t) - \alpha D_{KL}(q(\cdot|s_t) \;||\; \pi_\theta(\cdot|s_t)) \Big) \right] + \log p(\theta) \tag{2.30}$$

In eq. (2.30), $p(\cdot|q)$ is an auxiliary distribution over trajectories and approximates the true trajectory distribution induced by π (eq. (2.11)). The lower bound is the objective that is optimized with the EM algorithm, proposed in [AST$^+$18]. The E-step optimizes J with respect to q, while the M-step updates the parameterized policy π by regression on the trajectories generated in the E-step.

MPO is a stable and efficient off-policy algorithm that achieves state-of-the-art performance. Moreover, it poses an interesting alternative to common RL approaches by casting control as inference. This motivated the decision to include this algorithm as a baseline.

CHAPTER 3

Related Work

In recent years, deep RL gained a lot of attention in the field of intelligent robot systems and has been applied in many subdomains, such as perception, navigation, and control. In this chapter, we discuss important research progress that has been in the past. We organize this chapter into three sections. First, we discuss the current state of the art in general end-to-end robot control using model-free and model-based RL algorithms. Then we focus this discussion on the state of the art in autonomous racing and traditional approaches to this domain. Finally, we investigate recent advances in RL for autonomous racing.

3.1 Reinforcement Learning for Robot Control

Traditionally, many approaches in robot control have their roots in classical control theory and optimal control. However, robot control has been an active area of research for RL ever since. Many approaches from RL are closely connected to well-known planning algorithms and learning-based control. These types of algorithms are mostly known as model-based RL approaches, but also model-free methods found their way into robot control and deliver state-of-the-art performance. In the following, we outline important literature in the field of RL in the context of robotics and its connections to learning-based control.

Model-free RL for Robotics. Continuous action and state spaces are natural for robot systems. Early approaches of RL for robot control had to apply techniques to reduce the search space for such agents [KBP13]. Common techniques were action- or state-space discretization [MM02, vH12] or function approximation for dynamic programming [Gor95, TR96]. Before the advent of deep learning, function approximation mostly relied on hand-crafted features, not reaching the representational capacity of current deep learning models.

Modern deep RL algorithms usually employ deep neural networks as function approximators. Starting from standard policy gradient algorithms, numerous improved model-free actor-critic algorithms emerged and consequently pushed the state-of-art, especially in simulated robot environments, such as the *DeepMind Control Suite* [TDM$^+$18]. Algorithms, such as TRPO [SLA$^+$15], PPO [SWD$^+$17] and ACKTR [WML$^+$17] improved the learning stability for robot control and, under certain assumptions, even promise monotonic improvement guarantees. Off-policy algorithms, such as DDPG [LHP$^+$15] and its successors [BMHB$^+$18, FvHM18], improved the sample efficiency of RL, which is a crucial aspect for real-world robot systems. However, often model-free algorithms are extremely sensitive to hyperparameters, especially on real-world hardware [MKV$^+$18]. Algorithms such as MPO [AST$^+$18] and SAC [HZAL18] provide relative robustness to hyperparameter configurations.

Model-based RL for Robotics. In contrast to model-free approaches, model-based RL algorithms can leverage a given or learned dynamics model to improve performance. Model-based RL is an umbrella term for approaches that have access to a model of the robot or its environment. Within this family of algorithms, different approaches exist, such as Dyna-style approaches [Sut91], which learn a model to generate synthetic data to train policies, or approaches that use a learned model to plan into the future [HWMZ20]. Early model-based RL algorithms use Gaussian Processes to obtain low dimensional dynamics models from data [RK03, KKFH07]. PILCO is a prominent representative of models that use Gaussian processes to generate training data for policy search [DR11]. Other approaches use locally fitted models, which are simpler but do not provide global information [TL05, LA14]. The approaches mentioned so far use state-space representations that are low-dimensional, such as joint positions and velocities. However, some approaches are able to learn dynamics models for planning or policy search from high-dimensional observations, such as images [WSBR15, LFDA16]. A recent branch of work on model-based RL addresses world-model algorithms [HS18]. Such models learn an abstract state-space representation instead of a known state-space formulation from observations only. These latent states are then used to learn a latent dynamics model. This abstract model of the world can be used to do planning [HLF$^+$19] or train a model-free agent in latent space [ZVS$^+$19, HLBN20]. Furthermore, having a latent world model allows algorithms to backpropagate through the model, which allows more informative and less noisy parameter updates [WBC$^+$19, HLBN20].

Deploying RL to Real Robots. Many algorithms demonstrated superior performance in simulation. However, when deploying such agents to real-world robots, many challenges remain. Off-policy algorithms, such as SAC, improved the sample efficiency to a degree where learning directly on the robot can be feasible [HPZ$^+$18, HZH$^+$19]. Also, model-based algorithms can leverage data generated from real hardware [DR11, LFDA16]. However, when it is too expensive to obtain training data from hardware, simulated robot models are required to generate sufficient amounts of data. Deploying agents that are trained in simulation to real-world robots poses several challenges, due to the divergence of simulation and reality [AOS$^+$16, ZQW20]. Domain randomization is a simple approach to overcome the gap between simulated and real robots by randomizing properties of either the robot dynamics [RGLR16, PAZA18] or the simulated environment [TFR$^+$17, SL17] during training.

3.2 Autonomous Racing

Compared to urban driving, racing is a much more constrained and predictable task for autonomous agents. However, this reduced complexity allows concentrating research on autonomously driving cars in extreme conditions. Autonomous racing challenges, such as the Formula Student Driverless[1] or the F1TENTH Competitions[2] gave rise to research on autonomously racing vehicles.

State of the Art. Currently, racing competitions are dominated by teams that have highly complex and carefully designed computation pipelines. Autonomous racing systems, such as the AMZ Driverless, typically have dedicated modules for perception, planning, motion control and state estimation [KVR$^+$20, DDVG19, VBL$^+$20]. Depending on the competition, vehicles have access to a variety of sensors and use state estimation and mapping techniques to obtain a map, if not given [VHR$^+$18]. Racing line planning and optimization is an important step towards fast lap times. Often, racing line planning is framed as an optimization problem, with constraints based on vehicle dynamics [LB14] and road conditions [CWHL19]. To follow a precomputed racing trajectory, feedback control is often the tool at hand. For instance, Model Predictive Control (MPC) combined with sophisticated motion models allow accurate predictions and efficient online optimization [VBL$^+$20] and can be combined with online learning methods [KHLZ19, RB20].

RL for Autonomous Racing. While there exists a large body of research in the field of autonomous racing, there does not exist much work applying RL to learn racing tasks end-to-end. In [FSK$^+$21], the authors apply SAC to learn racing behavior in a video game. Similar to our approach, they use LiDAR measurements as inputs to the algorithm. Additionally, they also provide state variables, such as vehicle acceleration and velocity. Furthermore, they also propose a progress-based reward but penalize crashes with high kinetic energy. An approach that is capable of learning to race from images only is proposed in [JdT$^+$18]. The authors applied Asynchronous Advantage Actor-Critic (A3C) combined with Convolutional Neural Networks (CNNs) to extract features from images. The proposed reward function rewards agents that stay close to the centerline of the track.

[1]https://www.formulastudent.de/teams/fsd/
[2]https://f1tenth.org/race.html

CHAPTER 4

Approach

Based on a novel world-model algorithm, Dreamer [HLBN20], we designed an algorithm that is capable of learning to race on difficult tracks. The perception and control pipeline of our approach does not rely on any engineered modules and is learned purely from raw data. In the following, we will briefly introduce the original algorithm and subsequently describe the contributions to adapt it to the domain of racing. Lastly, the reward formulation is presented.

4.1 Introduction

As a world-model algorithm, Dreamer learns a compact description of the MDP in which the agent is embedded. This description is used to generate imaginary roll-outs to train a policy without having to collect tremendous amounts of observations, compared to model-free algorithms. At a high level, Dreamer alternates between phases of training its dynamics model, optimizing a policy acting in the abstract state-space and collecting data from its environment.

4.1.1 Dynamics Model

The world model in Dreamer is represented by an Recurrent State Space Model (RSSM), originally introduced in [HLF+19]. The RSSM consists of two components: A representation model p_θ and a generative model q_θ. States in RSSMs have a stochastic part, sampled from p_θ and q_θ respectively, and a deterministic part, which is computed by an Long Short-Term Memory (LSTM) network [HS97]. Formally, the RSSM consists of probabilistic models to encode observations into latent states and to predict the latent transition dynamics:

$$\begin{aligned}
\text{Representation model:} \quad & p_\theta(s_t|s_{t-1}, a_{t-1}, o_t) && (4.1)\\
\text{Observation model:} \quad & q_\theta(o_t|s_t) && (4.2)\\
\text{Reward model:} \quad & q_\theta(r_t|s_t) && (4.3)\\
\text{Transition model:} \quad & q_\theta(s_t|s_{t-1}, a_{t-1}) && (4.4)
\end{aligned}$$

The representation model encodes observations and previous states and actions into successor states, as shown in eq. (4.1). To predict latent states and rewards, Dreamer learns a generative model q_θ from data. This generative model predicts states and rewards just from latent states, depicted in eqs. (4.3) and (4.4). The observation model defined in eq. (4.2) is just used to provide a learning signal and not used during inference.

$$\begin{aligned}
J_{\text{rec}}(B) &= \frac{1}{|B|}\sum_{i=0}^{|B|}\sum_{t=0}^{H} J_{i,t}^{O} + J_{i,t}^{R} + J_{i,t}^{D} && (4.5)\\
J_{i,t}^{O} &= \log q_\theta(o_{i,t}|s_{i,t}) && (4.6)\\
J_{i,t}^{R} &= \log q_\theta(r_{i,t}|s_{i,t}) && (4.7)\\
J_{i,t}^{D} &= -\beta\, D_{KL}(p_\theta(s_{i,t}|s_{i,t-1}, a_{i,t-1}, o_{i,t}) \;||\; q_\theta(s_{i,t}|s_{i,t-1}, a_{i,t-1})) && (4.8)
\end{aligned}$$

The architecture of the RSSM is based on a Variational Autoencoder [KW14]. The information bottleneck, through which observations are passed, represents the latent state-space. Thus, the RSSM is trained by maximizing the ELBO. The ELBO consists of the terms denoted in eqs. (4.6), (4.7) and (4.8). The observation and reward log-likelihoods are represented by $J_{i,t}^{O}$ and $J_{i,t}^{R}$, respectively. The joint formulation of eqs. (4.6) and (4.7) provides a Maximum Likelihood estimate while eq. (4.8) serves as a regularizer and aims to minimize the information gain of using observations rather than just the latent transition model.

4.1.2 Latent Policy Optimization

Leveraging the compact state representation, Dreamer trains a policy in its latent state-space. The algorithm makes use of the action model q_ϕ, which produces an action distribution for a given state and a value function q_ψ:

$$\begin{aligned}
\text{Action model:} \quad & q_\phi(a_t|s_t) && (4.9)\\
\text{Value model:} \quad & q_\psi(v_t|s_t) && (4.10)
\end{aligned}$$

To predict the value of a state, dreamer uses an exponentially weighted average of different bootstrap estimates, denoted as V_λ^τ. Note that we explicitly express the dependence of V_λ^τ on τ, since the value estimate needs access to the whole trajectory. In [HLBN20], this notation is omitted for brevity. For more details, please refer to [HLBN20] and [SB18].

The training objectives for the action and value model are formulated as:

$$J(\phi) = \mathbb{E}_{\tau \sim q_\psi, q_\phi} \sum_{s \in \tau} V_\lambda^\tau(s) \tag{4.11}$$

$$J(\psi) = \mathbb{E}_{\tau \sim q_\psi, q_\phi} \sum_{s \in \tau} \frac{1}{2} \left\| v_\psi(s) - V_\lambda^\tau(s) \right\|^2 \tag{4.12}$$

When Equation (4.11) is maximized with respect to parameters ϕ, Dreamer increases the likelihood of trajectories τ that result in high value estimates. Note that this formulation is different from the basic policy gradient formulation, since Dreamer directly optimizes the values with respect to the policy parameters. In model-free algorithms, this is not possible due to the absence of a differentiable transition model. This aspect of Dreamer is also one of its main strengths. The gradient flow through the dynamics model allows more informed parameter updates, because the effect of actions on latent trajectories is directly observable and thus less noisy than typical policy gradient or actor-critic methods. The value model is fitted to the exponentially weighted average of values and returns, as shown in eq. (4.12).

4.1.3 Algorithm

Algorithm 4.1 depicts the training process of Dreamer. In lines 1 and 2, the parameters of the neural networks are initialized and dataset D is prefilled with random trajectories. Then the training loop runs alternating update- and data-collection steps. In lines 5 and 6, the algorithm samples a batch B of sequences of fixed length and optimizes the reconstruction loss defined in eq. (4.5). Starting from the states computed from B, Dreamer does a policy rollout over a fixed horizon in imagination using its world model in line 7. The resulting trajectories τ are then used to improve the action and value model, as depicted in lines 8 and 9. In the data-collection phase (line 11), Dreamer interacts with its environment by running an exploration policy. This policy adds exploration noise to the actions sampled from the action model. The resulting episodes are then added to the dataset.

4.2 Racing Dreamer

In order to apply Dreamer to racing tasks, we changed parts of its architecture and optimization procedure. First, since the observations are LiDAR scans, we replaced the CNN in the original Dreamer implementation with an Multi-Layer Perceptron (MLP). We did some experiments with 1D-CNNs, but results showed that MLPs are more suitable for this task. Most importantly, we introduced two reconstruction loss formulations to train the dynamics model. Additionally, we formulated several reward functions, tailored to the racing task.

Algorithm 4.1: Original Dreamer Algorithm [HLBN20]

```
1  Initialize parameters: $\theta$ (dynamics), $\phi$ (action model), $\psi$ (value model)
2  Initialize dataset with $S$ random episodes.
3  while not converged do
4      for $C$ update steps do
5          Sample random sequences $B$ from dataset.
6          Fit dynamics model: $\theta \leftarrow \theta + \alpha \nabla_\theta J(B)$ (see eq. (4.5)).
7          Compute and collect imaginary trajectories using the dynamics model and
             the latent policy.
8          Update action model: $\phi \leftarrow \phi + \alpha \nabla_\phi J(\phi)$ (see eq. (4.11)).
9          Update value model: $\psi \leftarrow \psi - \alpha \nabla_\psi J(\psi)$ (see eq. (4.12)).
10     end
11     for $T$ episodes do
12         Run exploration policy $\pi_\phi$ in environment, using action model.
13         Add episodes to dataset.
14     end
15 end
```

4.2.1 Reconstruction Losses

Instead of image reconstruction, we experimented with two different reconstruction losses. First, we adopted the observation reconstruction to LiDAR scans, which we call *Distance Reconstruction*. Second, we reconstruct local patches of the occupancy grid map from the latent state representation (*Occupancy Reconstruction*). Visualizations of the resulting reconstructions are depicted in fig. 4.1. In the following, we denote Dreamer using distance reconstruction as Dreamer-Distance and Dreamer using occupancy map reconstruction as Dreamer-Occupancy.

Distance Reconstruction. Analogous to the original implementation, we obtain a reconstruction loss by reconstructing the distance measurements obtained by LiDAR scans. The reconstruction is sampled from a normal distribution, whose parameters are obtained by an MLP, that maps latent states to the mean and variance of the reconstructed measurement. The distribution is an independent normal distribution and has the same number of dimensions as the LiDAR scan, hence has a scalar normal-distribution for each ray.

Occupancy Reconstruction. While plain observation reconstruction is a very generic approach and works well on unseen environments, we also believe that LiDAR scans do not contain rich amounts of information about the vehicle's environment. Therefore, we developed another reconstruction loss, based on reconstructing local maps in the vicinity of the agent. Dreamer still processes LiDAR observations, but instead of reconstructing the scan, the observation model outputs a bitmap representing an occupancy grid

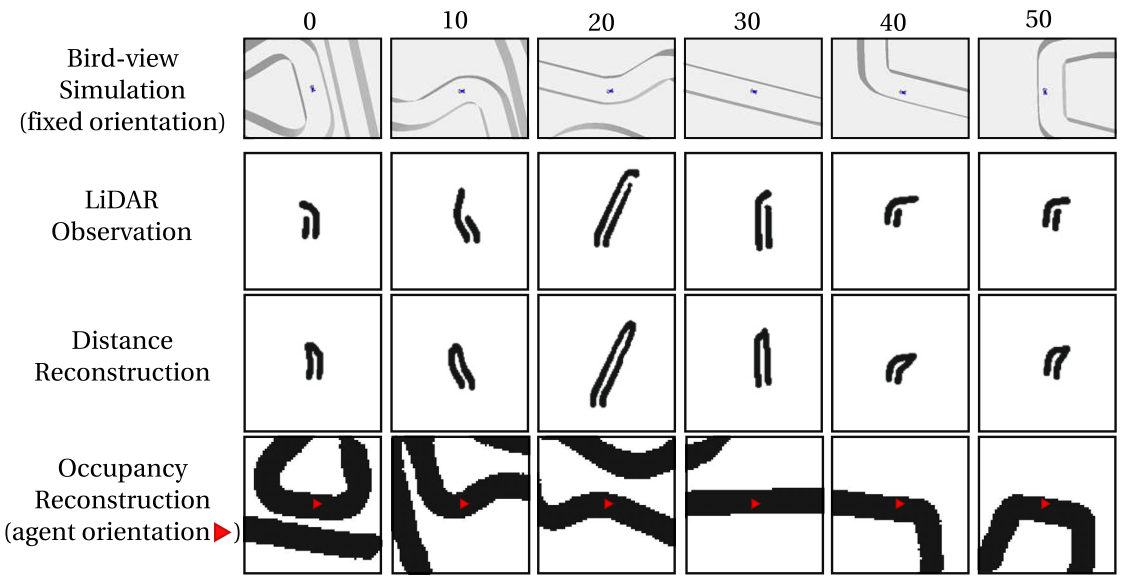

0
10
20
30
40
50
Bird-view Simulation (fixed orientation)
LiDAR Observation
Distance Reconstruction
Occupancy Reconstruction (agent orientation ▶)

Figure 4.1: Reconstructions of LiDAR scans and occupancy maps over 50 time steps. Observations are in agent coordinates. **Row 1:** bird-view of the race car in simulation, **Row 2:** LiDAR scan in race car coordinates, **Row 3:** reconstructed LiDAR scan, **Row 4:** reconstructed local occupancy map

map centered around the agent. The latent state is processed by multiple layers of deconvolutional layers which output the mean of a Bernoulli distribution for each pixel in the grid map. The bitmap is then sampled from this distribution and the loss is computed by comparing the output with a patch of the true map.
Figure 4.1 shows an example of the map reconstruction output. The bottom row depicts the reconstructed grid map centered around the agent. Note that the map is rotated by 90 degrees, due to coordinate system differences. One can see, that the agent learned to accurately reconstruct the map of its environment. Thus, we expect that this approach results in better performance because the reconstruction loss might be more informative due to the association of LiDAR scans and the map. Indeed, this approach yields better results, but also has some drawbacks, such as overfitting to the training environment. For more detailed results we refer to chapter 4.

4.2.2 Reward Formulation

While standard benchmarks, such as *MuJoCo* [1], have well-designed reward functions, we had to create a suitable training signal for the racing task. Our goal was to find a well-defined reward function that steers the agent to learn to race while encoding as little bias as possible into it. The reward functions proposed in this section have access to the true state of the simulation. This includes ground truth pose, velocity, collisions,

[1]`https://gym.openai.com/envs/#mujoco`

and other important environmental information. We experimented with a number of different approaches before agreeing on a progress-based formulation. In the following, we will present discarded approaches and explain the progress-based reward in detail. Please note that, when we refer to a state s_t in this subsection, we mean the state of the simulation and not the latent state representation of Dreamer.

Discarded Reward Functions. We designed multiple reward functions before we arrived at the progress-based formulation. Our first approach was to just penalize the time it takes the agent to complete a lap:

$$R(s_t, a_t) = \begin{cases} -t, & \text{if } t \text{ is terminal} \\ 0, & \text{otherwise} \end{cases} \tag{4.13}$$

However, using this reward definition, the agent was not able to learn to complete a lap. We believe, that this reward formulation is too sparse in our domain, as the agent needs to complete a whole lap before receiving any feedback. We tried variations of this approach by discretizing the track into sections and minimizing the time it took the agent to complete sections of the track. However, this also did not lead to successful learning. We suspect that the number of time steps between rewards is still too large, since the simulation step frequency of 100 Hz results in hundreds of time steps between two sections. An alternative formulation of the reward function defined in eq. (4.13) is to penalize the agent for each time step by a small constant c, until it reaches the finish line:

$$R(s_t, a_t) = -c. \tag{4.14}$$

However, this did not resolve the problem of reward sparsity and did not yield good results. Another reward function we designed, aimed to maximize velocity and penalizing crashes:

$$R(s_t, a_t) = \begin{cases} -1, & \text{if agent is in collision} \\ s_{t,v} - |a_{t,\phi} - a_{t-1,\phi}|, & \text{otherwise} \end{cases} \tag{4.15}$$

In addition to maximizing the velocity, eq. (4.15) also minimizes the magnitude of the change of the steering control ϕ between two consecutive time steps.

Progress-based Rewards. Based on the grid maps of the tracks, we computed additional semantic maps. One such map, which we name *progress map*, shows the normalized distance from the starting line for each position on the track. Thus, we have an accurate estimate of the progress an agent made in one lap. Figure 4.2 shows a visualization of the progress maps used during training. The progress is gradually increasing, visualized as a color gradient from lighter to darker grayscales.

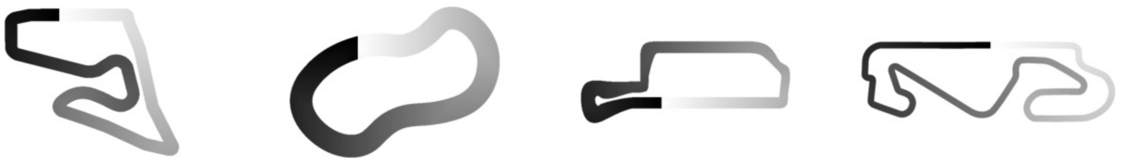


Figure 4.2: Progress maps of Austria, Columbia, Treitlstrasse and Barcelona.

Based on the progress maps, we designed a reward function that measures the difference of the progress values at distinct time steps. We refer to this reward as *progress-based reward*. The basic idea is to maximize the covered progress within a small amount of time:

$$R(s_t, a_t) = \begin{cases} -C, & \text{if in collision} \\ |s_{t,p} - s_{t-c,p}|, & \text{otherwise} \end{cases} \tag{4.16}$$

In eq. (4.16), $s_{t,p}$ denotes the progress value at time t and c is a hyperparameter, scaling the time window to compute the difference. In our experiments, we set $c = 1$. If the agent is involved in a collision with the wall or any other agent, the agent receives a penalty C and the episode is terminated. The idea behind this formulation is that, if an agent manages to maximize the progress within a given time window on small, localized parts of a track, it will lead to a globally optimal behavior. Furthermore, this reward definition provides a dense learning signal, as it gives feedback at any time during training.

CHAPTER 5

Methodology

This chapter introduces the setup and training modalities we defined to train and evaluate agents. First, we present the simulation environment we developed to train agents. Subsequently, we introduce the robot platform which serves as a real-world scenario to test the trained models. Then we focus on the racing tracks we designed to challenge autonomous racing agents. Lastly, we lay out the training regime for the racing agents and the algorithm implementations.

5.1 Simulation Environment

We make use of a simulated environment to train and test agents. In the following, we motivate and explain the simulation platform. Furthermore, we discuss the advantages and challenges that come hand in hand when training RL agents in simulated environments.

5.1.1 Motivation

In contrast to supervised learning, the distribution from which training data for RL algorithms is drawn is usually not stationary over time. Therefore, training RL agents often requires immense amounts of training data [BRW$^+$19]. Generating this amount of experience on the real robot is often not feasible. For instance, training autonomous race cars would require manual resets of the cars which results in an extreme increase in training time. Simulations can drastically decrease the time required to generate samples because they allow to run models faster than real-time. Furthermore, it is possible to scale simulations horizontally by distributing the workload over many servers. For these reasons, RL agents are usually trained in simulated environments.
Besides throughput considerations, safety is another aspect that has to be considered during training. In contrast to established solutions which often have access to predictive models and prior knowledge, such as safety constraints, RL algorithms learn by exploring

the state-space. In safety-critical domains, such as autonomous driving, or in domains where uninformed controls lead to high costs or even damage, it is not possible to learn a policy directly on the hardware. Instead, policies are learned in more or less accurate simulations, where failures do not have severe consequences. One illustrative example is an autonomous driving agent that aims to learn an optimal policy that allows safe navigation in urban traffic. To learn that crashing into other traffic participants is highly discouraged, it would have to experience such situations at least once, but in most cases, many times.

5.1.2 Challenges

Training RL agents in simulation does not come without challenges. The distribution from which simulated sensor data is generated might be different to the distribution of the real-world sensor data. This phenomenon is also known as *distribution shift*. Typically, RL algorithms tend to be sensitive to this type of divergence between simulation and reality [AOS+16]. There are approaches, such as domain randomization, to make RL algorithms less sensitive to distribution shift [RGLR16, TFR+17, PAZA18]. However, it remains a key challenge to be solved. Another issue that occurs when training agents in simulated environments is *reward hacking* [AOS+16]. This type of problem occurs because RL agents tend to exploit inaccuracies in simulators or reward function definitions, if they can maximize the reward signal by doing so. Often, the policy obtained by reward hacking exhibits undesirable properties and does not reflect the true intent of the objective.

5.1.3 Racecar Gym

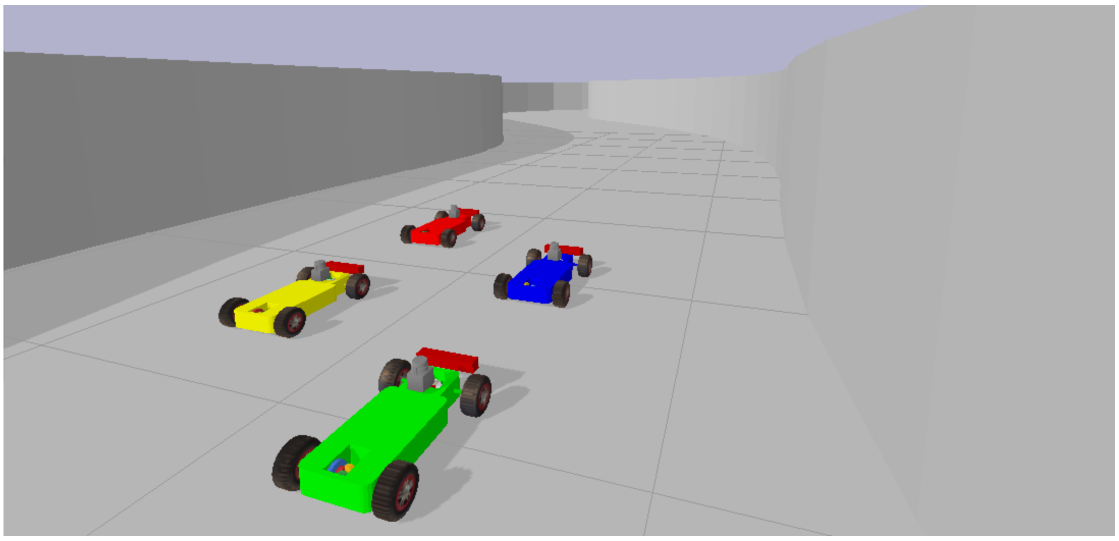
We developed a custom race car simulation platform to train our agents. Figure 5.1 shows an example of a rendered scene with multiple agents. The simulation environment is based on *OpenAI Gym* [BCP+16], a library that defines a simple interface specification to model MDP's, and *PyBullet* [CB19], a python interface to the Bullet simulation engine. In the following, we will describe the design goals we had in mind when developing the simulation envrionment and which features are available. The software is open-source and can be found on GitHub[1].

Multi-Agent Racing Scenarios. Although we focus on single-agent racing tasks in this work, we designed the simulation environment to support multi-agent scenarios to enable further research work in this direction. Thus, agents are identified by IDs and can be configured to either solve cooperative tasks or compete against each other. Furthermore, it is possible to create scenarios in which teams of agents solve contrasting tasks.

Configurability. An important aspect of our simulation environment is its ease of configuration. It is possible to customize various aspects of scenarios, such as the track,

[1] `https://github.com/axelbr/racecar_gym.git`

Figure 5.1: Rendered scene of a scenario simulated in *Racecar Gym*.



race car model, available sensors, and tasks assigned to the agents. Listing 1 shows an example of a configuration file for a multi-agent scenario. One can choose from multiple tracks of varying difficulty (see section 5.3). Agents are characterized by the vehicle they control and the task that is assigned to them. In multi-agent scenarios, agents are identified by an ID. Vehicles can be configured by changing their underlying robot model and by attaching various sensors to them. Examples can be found in listing 1. Furthermore, agents are tasked with configurable, parametrizable objectives.

Extendability. We aimed to implement an environment, that is easy to extend in various ways. In order to easily switch reward functions, it is possible to create and register custom tasks and configure them in scenario specifications. Furthermore, one can design new tracks and vehicle models and plug them into a scenario specification. The models of the tracks can be created with any 3D modeling software. Furthermore, we also provide a prototype to automatically generate track models just from occupancy grid maps that are provided in the map-server format[2].

Realistic Physics. To accurately simulate the dynamics of the robot platform, we chose the physics simulation software *Bullet* [CB19]. The car model is a rigid body system based on the URDF model proposed in [BB20]. URDF is a declarative modeling language for robot systems and is based on XML. The vehicle model we use allows the configuration of geometric properties, transmission reduction proportions, and friction coefficients. The model is actuated by applying force to the steering and acceleration

[2]http://wiki.ros.org/map_server

```yaml
world:
  name: austria
agents:
  - id: A
    vehicle:
      name: racecar
      sensors: [lidar, pose, velocity, acceleration]
      color: blue
    task:
      task_name: maximize_progress
      params: {laps: 1, time_limit: 120.0}
  - id: B
    vehicle:
      name: racecar
      sensors: [rgb_camera, pose]
      color: red
    task:
      task_name: maximize_progress
      params: {laps: 1, time_limit: 120.0}
```

Listing 1: Example scenario specification.

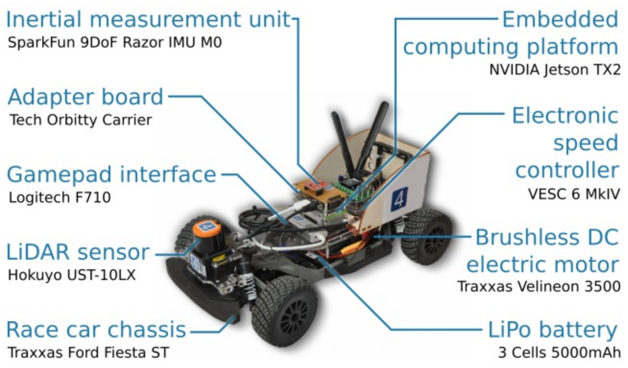
joints. Bullet can simulate a broad set of sensory inputs, such as LiDAR sensors and RGB cameras.

5.2 Robot Platform

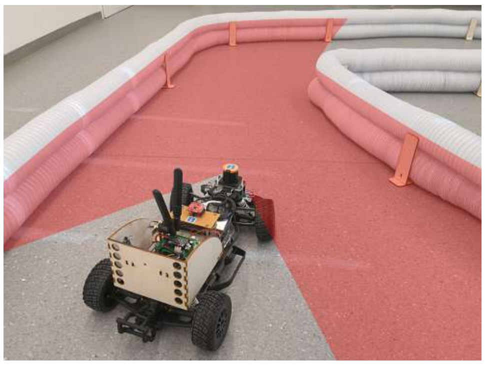
In our experiments, we examine the performance of agents on small-scaled race cars after they have been trained in simulation. This section outlines the hardware setup of the robot and gives an overview of the software architecture we use to conduct experiments on the hardware. The evaluation platform is based on the F1TENTH race car series [OSA+19].

5.2.1 Hardware Setup

The autonomous race car is based on an off-the-shelf 1/10 scaled model race car chassis, with a Traxxas Velineon 3351R brushless DC electric motor. The motor is controlled by an electronic speed controller (ESC). We use a VESC 6 MkIV ESC for this purpose. For onboard computational tasks, such as inference, we use an NVIDIA Jetson TX2 embedded computing board. It has 6 ARM-CPU cores, an integrated GPU with 256 cores for fast processing tasks and 8GB of memory. Furthermore, we use an Orbitty Carrier Board to enhance the IO capabilities of the NVIDIA Jetson TX2. Sensory inputs are obtained by a laser range sensor and an intertia measurement unit (IMU). The range sensor is a Hokuyo UST-10LX 2D LiDAR sensor. It generates 1081 range measurements over an angle of 270° per scan, providing an angular resolution of 0.25°. Range measurements have a maximum detection range of 30 meters with a high accuracy range of 10 meters [Hok20]. Figure 5.2b shows an illustration of a LiDAR scan on a race track. The IMU is a 9DoF Razor IMU M0. It provides a 3-axis accelerometer, gyroscope and compass [TDK17]. Figure 5.2a shows an image of the car with annoted components.



(a) Hardware components of the race car.



(b) Illustrated LiDAR scan.

Figure 5.2: Race car platform used for evaluation.

5.2.2 Software architecture

The software system that controls the race car is based on Robot Operating System (ROS), a message-oriented middleware designed for robotics applications [QCG+09]. ROS-based applications run in a distributed fashion, organized into nodes that are communicating via messaging APIs. The architecture of our system is shown in fig. 5.3. ROS-nodes are marked in green, hardware components in yellow and the trained RL agent is a python program, marked in purple.
The embedded computing board, an NVIDIA Jetson TX2, runs the core infrastructure which is necessary to communicate with sensors and the motor controllers. It is connected to the LiDAR sensor via an ethernet interface and to the VESC motor controller via a USB interface. Additionally, we attached a wireless gamepad for manual control and emergency stops. To receive sensor data and send commands to the motor controller, we run two ROS nodes that connect the hardware to our system. The *URG node* reads LiDAR scans from the ethernet interface and wraps the data in the appropriate message format. The *VESC node* receives motor commands from the agent and applies a low-pass filter to protect the hardware before sending it to the Electronic Speed Control (ESC).
For the deployment of trained agents, we use Docker containers. These containers communicate with the native ROS system via TCP/UDP and are independent of the ROS version. For instance, to deploy trained RL agents, we use a newer version of ROS in the docker container than it is available on the base system.

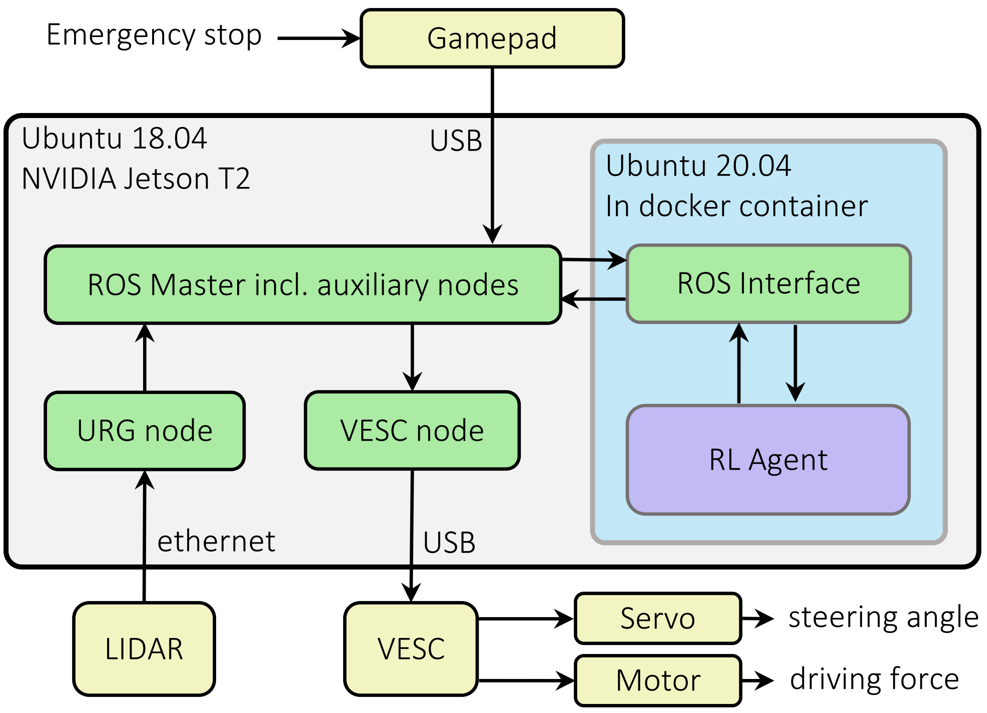


Figure 5.3: Software architecture diagram of the race car.

5.3 Race Tracks

For training and evaluating the racing performance of the agents, we use a selection of race tracks of varying difficulty. Figure 5.4 shows the layout of the tracks. All of the tracks are available as simulated environment models and can be downloaded from the repository. The simulated tracks are 3D models which can be loaded into the physics engine. We created the tracks by extruding vector graphics of the track layouts using Blender[3]. However, any other CAD software could be used potentially. For Columbia, Austria, and Barcelona we use existing track layouts that are available online[4]. For Treitlstrasse, which is our real-world evaluation track, we computed an occupancy grid map from data that we collected by manually driving the car on the track. From this data, we could compute a map with existing Self-Localization and Mapping (SLAM) packages[5] available for mobile robots using ROS.

[3]https://www.blender.org/
[4]https://github.com/CPS-TUWien/racing_dreamer/tree/main/docs/maps
[5]https://google-cartographer.readthedocs.io/en/latest/

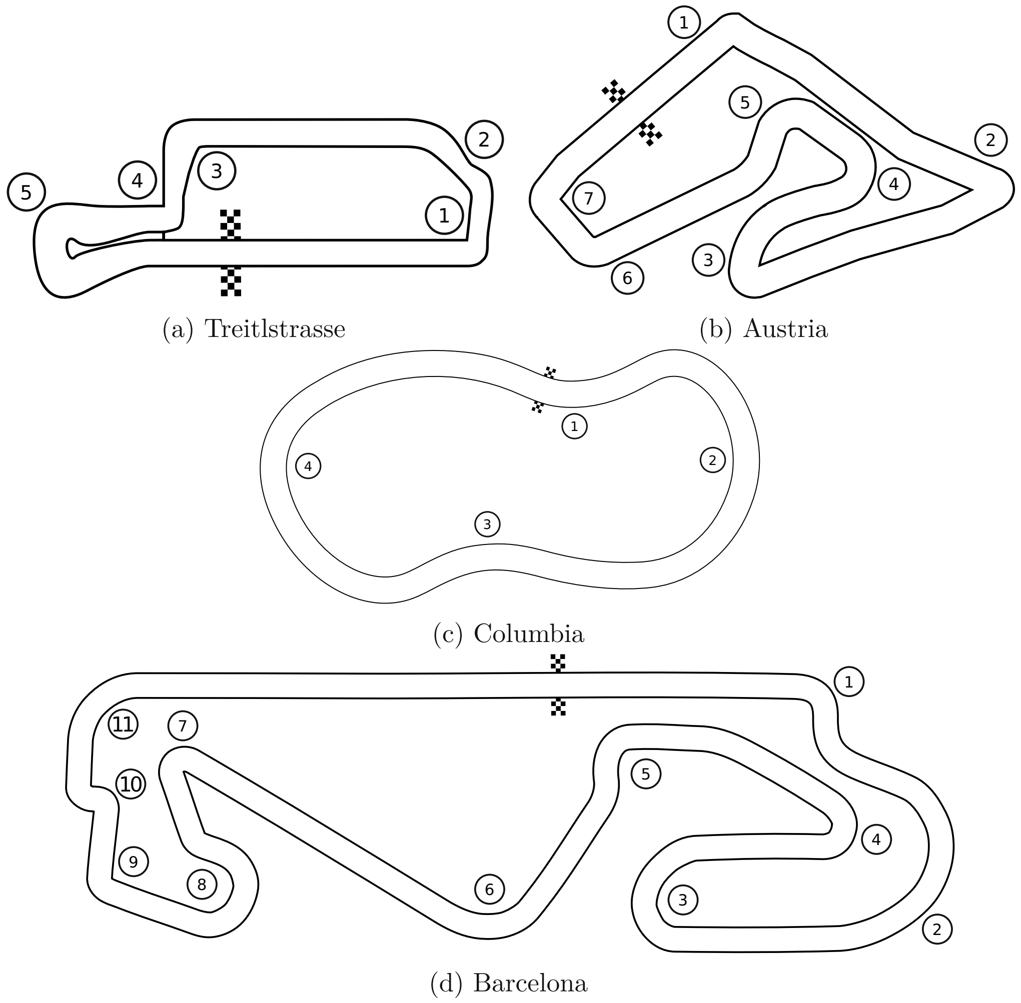


Figure 5.4: Race tracks with marked start line and numbered turns.

Characteristics

To compare the results of racing agents across different tracks, we computed characteristics for each track. Table 5.1 shows the mean track width and its standard deviation, track length, and minimal curve radius for each map (as proposed in [BCMS08] and [LL01]). The mean and standard deviation of the width is computed by extracting the center-line for a map and measuring the width at each position along this line. Table 5.1 shows, that the tracks Austria, Columbia and Barcelona all have a comparatively uniform width in contrast to Treitlstrasse. The widest track is Columbia, with a mean track width of 3.44 meters, the longest track is by far Barcelona, with a length of more than 200 meters. Additionally, we computed the curve radius by finding the largest circle that fits the outside of a curve, for each curve. This metric gives an estimate of the difficulty of curves, assuming tighter curves are harder to navigate.

Table 5.1: Track characteristics.

Track	Width (mean/std.)	Length	Min. Radius
Austria	1.83/0.04 m	79.45 m	2.78 m
Columbia	3.44/0.09 m	61.20 m	7.68 m
Treitlstrasse	1.36/0.25 m	51.65 m	3.55 m
Barcelona	1.82/0.10 m	201.00 m	2.98 m

Track Difficulty

Given the metrics in table 5.1, we claim that Columbia is the easiest track for racing, because of its large, uniform width and wide curves. This claim is also supported by the results we obtain in chapter 6. Based on this observation, we tested all our approaches first on Columbia before moving on to harder tracks, such as Austria. However, it is not as clear which track is the hardest to accomplish. The varying track width, narrow passages, and tight curves of Treitlstrasse are good indicators of its difficulty. However, we observed that all agents had also big troubles on Austria, because of its tight turns, especially the second turn. Therefore, we can name the easiest track, but we can not nominate the most difficult track without doubt. These considerations are important to interpret and relate the racing results across different tracks.

5.4 Implementation Details

In this section, we introduce the details of the training regime and discuss the implementation and configuration details we used to conduct experiments. Furthermore, we describe the hyperparameter optimization procedure which we conducted and subsequently the results. We open-sourced all implementations and configuration files on GitHub. The repository is located at `https://github.com/CPS-TUWien/racing_dreamer`.

5.4.1 Policy Optimization Procedure

Agents are trained in simulation on different tracks. Observations and actions are normalized to the intervals $[0, 1]$ and $[-1, 1]$, respectively. The model-free baseline algorithms are trained for 8 million environment steps in total. Dreamer agents are trained until they collected 2 million environment steps. The training loop alternates between training phases and evaluation runs in fixed intervals.

Training. We set the time horizon for a training episode to 20 seconds. Given a simulation time step of 0.01 seconds, one episode has a length of 2000 time steps. At the beginning of each episode, we place the agent on a random pose on the track, instead of starting from a fixed position to prevent overfitting to certain regions of the tracks. The pose consists of a uniformly sampled position from the map and a random heading direction. The heading is sampled from a distribution that is biased towards the intended racing direction. Figure 5.5 shows a visualization of 50 sampled starting poses on Austria. The objective function that is optimized is the progress-based reward proposed in chapter 4.

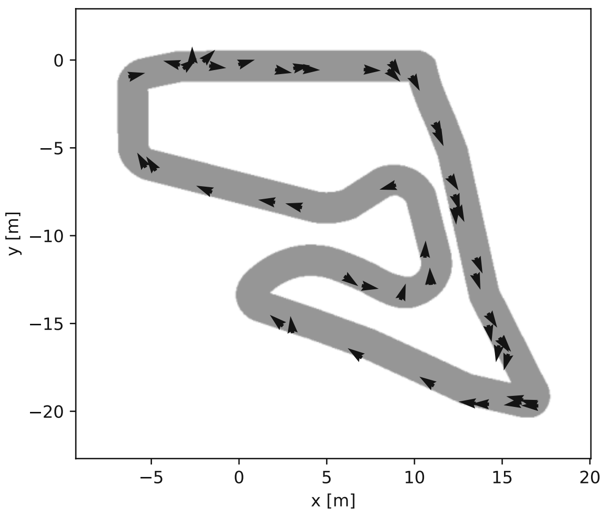


Figure 5.5: Randomly sampled starting poses.

Testing. We monitor the racing performance of agents during training at regular intervals. In contrast to training runs, we place the agents on fixed starting positions for each track so that we can compare the performance between multiple test runs. In order to compare the performance of agents that are trained with different objectives or parameters of the same objective (which would result in different amounts of rewards) we introduce a metric called Maximum Absolute Progress (MAP). MAP is defined as the maximum progress achieved on a map within a given time window. For our experiments, we choose a time window of 40 seconds and average the MAP over 5 consecutive trials. Furthermore, this metric allows us to compare trained agents with traditional approaches such as Follow-the-Gap (FTG; [SG12]), which does not optimize a reward function.

5.4.2 Model Architectures and Algorithm Implementations

In the following, we describe the model architectures and software implementations of the algorithms and approaches explained in chapters 2 and 4. We describe models that are realized as neural networks by the type and dimension of their layers and present their architecture in tables. For more details, such as reshaping and activation functions, please refer to the code repository.

Dreamer

The algorithmic contributions we described in chapter 4 (distance and map reconstruction) are implemented on top of the existing implementation of Dreamer [Haf19]. This implementation is written in the *Python* programming language and makes use of the hardware-accelerated tensor math library *Tensorflow*[6].
The transition and representation model are combined into the Recurrent State Space Model (RSSM), which is used as a latent forward model but also incorporates observations using the representation model, if available. As shown in table 5.2, the transition model consists of a fully connected layer with 200 neurons, followed by a GRU-Cell (Gated Recurrent Unit; [CvMG+14]) and another MLP. The outputs of the transition model are the mean $\mu \in \mathbb{R}^{30}$ and standard deviation $\sigma \in \mathbb{R}^{30}$ of a multivariate normal distribution from which the stochastic part of the next state is sampled. The GRU-cell outputs the deterministic part of the consecutive state. The total state dimension is thus 230. The representation model combines observations with state predictions using an MLP with a single layer.

Module	Layers	Dimensions
Transition Model	Dense Layer	200
	GRU-Cell	hidden state dim.: 200
	MLP, 1 hidden Layer	200/60
Representation Model	MLP, 1 hidden layer	200/60

Table 5.2: Architecture of the RSSM.

The architecture of the two reconstruction models presented in chapter 4 is shown in table 5.3. The distance reconstruction model is implemented as a two-layered MLP. The input is the latent state representation, a 230-dimensional vector, which is mapped to vectors $\mu, \sigma \in \mathbb{R}^{1080}$, the parameters of an independent multivariate normal distribution. This distribution represents the observation model $q_\theta(o_t|s_t)$.
The reconstruction of the occupancy grid map is realized by transforming the latent state to the parameter $p \in \mathbb{R}^{64\times 64}$ of a Bernoulli distribution, predicting the probability of a color (white for free, black for occupied) for each pixel. The mapping from a state $s \in \mathbb{R}^{230}$ to the output is achieved by transposed 2D convolutions, as shown in table 5.3. To learn and predict rewards in latent space, we use two fully connected layers. The output is the mean of a scalar normal distribution with fixed standard deviation $\sigma = 1$, modeling the reward signal.

The action and the value model are implemented as fully connected MLPs, as shown in table 5.4. The action model has 4 hidden layers, each of dimension 400. The outputs of the action model are the parameters $\mu, \sigma \in \mathbb{R}^2$ of a two-dimensional Gaussian, representing

[6] `https://www.tensorflow.org`

Module	Layers	Dimensions
Distance Reconstruction	MLP, 2 hidden layers	256/512/2160
Occupancy Reconstruction	Dense Layer	64
	ConvTransposed2D	channels: 32, width: 5, stride: 2
	ConvTransposed2D	channels: 16, width: 5, stride: 2
	ConvTransposed2D	channels: 8, width: 6, stride: 2
	ConvTransposed2D	channels: 1, width: 6, stride: 2
Reward Model	MLP, 2 hidden layers	400/400/1

Table 5.3: Architecture of reconstruction models.

the action distribution. The value model is a fully-connected MLP with 3 hidden layers that output the mean of a normal distribution with a fixed standard deviation $\sigma = 1$.

Module	Layers	Dimensions
Action Model	MLP, 4 hidden layers	4×400/4
Value Model	MLP, 3 hidden layers	3×400/1

Table 5.4: Architecture of the action and value module.

Baseline Algorithms

To compare the performance of Dreamer to other RL algorithms, we use open-source implementations of the model-free algorithms presented in chapter 2. For the MPO and D4PG agent, we used the implementations of the ACME framework [HSA+20], for SAC and PPO we chose the *Stable Baselines 3* library [HRE+18, RHE+19].
We kept the architectures and layer dimensions for policies and value functions close to the action and value model of Dreamer, as shown in table 5.4. The number of layers and the dimension of the policy networks for MPO, D4PG, PPO, and SAC are depicted in table 5.5. All policies consist of 4 hidden layers with 400 neurons each. The output is the action vector of dimension 2. Note that for MPO, SAC, and PPO, there is an additional linear layer to compute the mean and standard deviation of the action distribution. The value model is an MLP with 3 hidden layers of the same width.

Module	Layers	Dimensions
Policy Network	MLP, 4 hidden layers	4×400/2
Critic Network	MLP, 3 hidden layers	3×400/1

Table 5.5: Policy and critic network architectures for MPO, D4PG, PPO and SAC.

Table 5.6 shows the architecture of the LSTM-based PPO agent. The policy and the value network share a common recurrent observation network. The observation network consists of an MLP with a single hidden layer and an LSTM network with 256 cells. Thus, the policy and the critic network are reduced by one layer, to accommodate for the additional observation network.

Module	Layers	Dimensions
Shared Observation Network	MLP, 1 hidden layer LSTM, 256 cells	200/200 hidden state dim.: 200
Policy Network	MLP, 3 hidden layers	3×400/2
Critic Network	MLP, 2 hidden layers	2×400/1

Table 5.6: PPO-LSTM Architecture

5.4.3 Hyperparameter Tuning

We conducted a hyperparameter study to find better hyperparameters than the default parameters provided by the frameworks. As an optimization library, we used the *Optuna* hyperparameter optimization framework [ASY+19]. We defined the objective function of the search procedure to be the averaged MAP over 5 consecutive evaluation runs for 4000 time steps, reached after training for 1×10^6 time steps on Austria. For each agent, we evaluated 100 hyperparameter samples. The sampling process is governed by the Tree-Structured Parzen Estimator [BBBK11, BYC13] combined with Hyperband pruning [LJD+18]. The pruning algorithm allowed to prune unpromising hyperparameters early during training, which resulted in a significant reduction of training time.
The results showed that some configurations were able to match the results we obtained using the default parameters but did not exceed their performance. This led us to the conclusion, that the default parameter settings were already well suited for our task. We outline the selection of parameters that were tuned and the exact search spaces as well as the final values for the parameters in the Appendix.

CHAPTER 6

Results

In this chapter, we present the experiments we conducted and the results which we could obtain. If not stated otherwise, all experiments were conducted using the same training curriculum that we introduced in chapter 5. In our experiments, we compared the performance of Dreamer to state-of-the-art model-free algorithms, introduced in chapter 2. Thus, the baseline algorithms we choose to test against are D4PG, MPO, PPO (with and without LSTM) and SAC. Additionally, we also compared the racing performance to an agent using Follow the Gap (FTG), which is tuned per track.

We conducted three different experiments to assess the capabilities of Dreamer in the context of racing:

1. Comparison of the training performance and sample efficiency.
2. Evaluation of the generalization ability across multiple tracks.
3. Evaluation of the performance on our real-world test track.

6.1 Learning Performance Evaluation

We compared the learning curves and sample efficiency for all agents across multiple tracks. Figure 6.1 shows the learning curves for all agents on all tracks. We trained the baseline algorithms for 8 million time steps, Dreamer for 2 million time steps. The top row in fig. 6.1 shows the evolution of the average MAP metric over 5 runs during training for all model-free agents. To compare it with Dreamer, we show the best MAP reported during the training of Dreamer, visualized as a blue dashed horizontal bar. In the bottom row, we show the performance of Dreamer using the two reconstruction models proposed in chapter 4. The horizontal bars indicate the best MAP reported for every model-free algorithm.

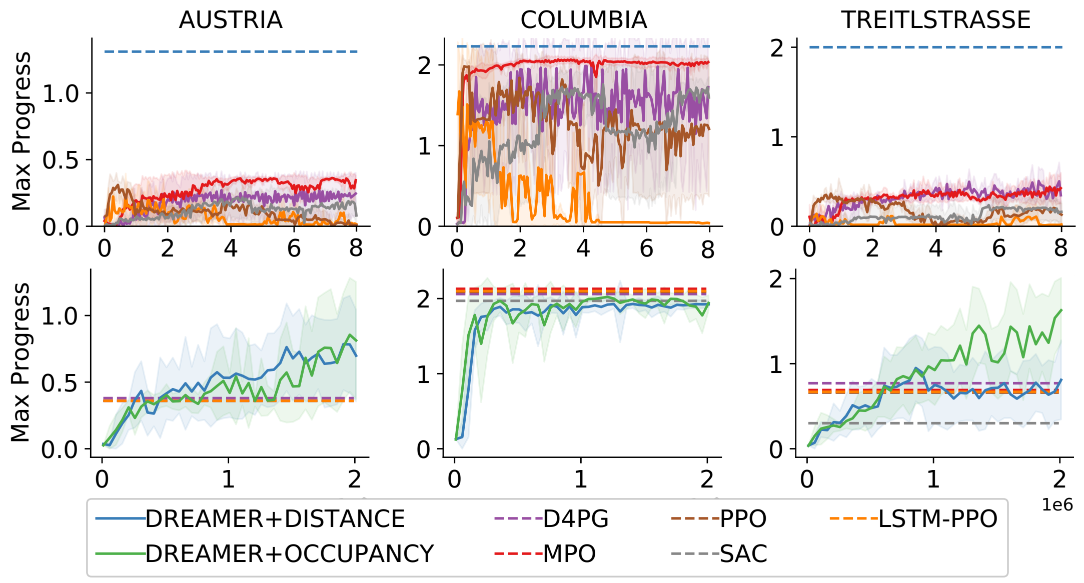


Figure 6.1: Learning curves for all agents averaged over 5 runs. Shadowed regions are standard deviations. **Top:** model-free learning curves over 8×10^6 steps, dashed line: best reported MAP of Dreamer. **Bottom:** Dreamer learning curves over 2×10^6 steps, dashed lines: best reported MAP of all model-free agents.

6.1.1 Performance Comparison

On Columbia, all algorithms were able to achieve a MAP of approximately 2 or more, which corresponds to driving two laps within 40 seconds. This performance is on par with agents trained with Dreamer. The plots show that the performance of agents using PPO and the LSTM version of PPO tend to collapse after achieving good results early in the training. From our results, we can observe that MPO achieves the most stable learning performance, while also maintaining high performance values. D4PG also achieved good results, exhibiting a high but stable variance in performance during training. The performance of agents trained with Dreamer is approximately the same and also did not collapse throughout the training process.
On more difficult tracks, such as Austria and Treitlstrasse, model-free algorithms had difficulties completing a single lap. Figure 6.1 shows, that no agent, that was trained with any of the baseline algorithms, achieved a MAP of 1 or more. The best performing algorithms, MPO and D4PG, achieved MAP values around 0.4 and 0.8 on Austria and Treitlstrasse, respectively. The plots indicate that on these two tracks, model-free agents get stuck in distinct parts of the tracks. After further investigation, we observed that the performance limits correspond to difficult parts of the tracks. Figure 6.2 shows the regions, where model-free agents were not able to drive without crashing or exceeding the time limit. On Austria, no baseline agent mastered the second turn (shown in fig. 6.2a), which is a less than 90 degree right turn. Even after biasing the sampling distribution of

the starting positions to regions just before this turn to increase the experience density in this region, no model-free agent was able to reliably get around this corner.
According to the results, the most difficult parts of Treitlstrasse are the two turns circled in red, shown in fig. 6.2b. We believe, the narrow passage and the sharp right turn in combination contribute the most to the difficulty of this part of the track. Although agents trained with Dreamer also have difficulties in these parts of the track, they can often overcome the passages after enough training. On average, agents trained with Dreamer on Treitlstrasse using distance reconstruction, perform as well as the best reported model-free results. However, if Dreamer is trained with map reconstruction loss, performance significantly increases and overcomes all other approaches. We believe, that this performance gain is achieved by overfitting to the map. This observation is also supported by the results reported in section 6.2.

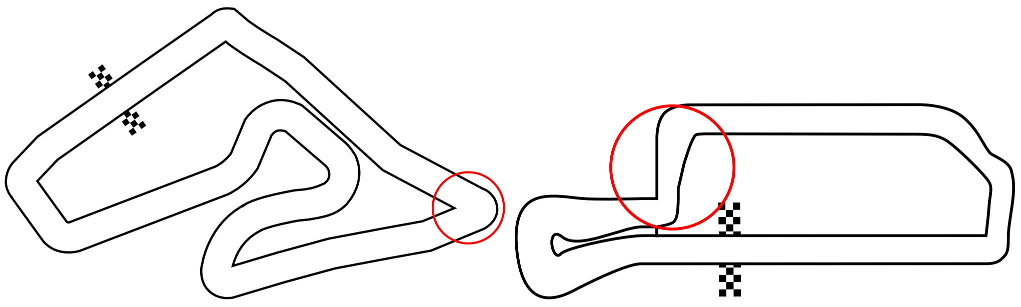

(a) Austria (b) Treitlstrasse

Figure 6.2: Difficult regions of the tracks.

6.1.2 Sample Efficiency and Computational Costs

In general, model-based RL algorithms are often more sample efficient than model-free RL algorithms. This general claim is supported by the results we obtained. Figure 6.3 compares the learning performance of Dreamer with the model-free baseline agents on Austria. Dreamer surpasses the performance of all other agents at a fraction of training samples, although all agents use roughly the same amount of parameters for their policy networks. This shows, that utilizing a learned environment model to generate latent experience does improve sample efficiency significantly. However, this advantage comes at a cost. The computational cost of Dreamer exceeds the computation requirements of the model-free agents by a significant portion. We observed that a full training run of Dreamer, which is 2 million time steps, takes approximately 48 hours. Training the model-free algorithms takes 8 to 12 hours for 8 million time steps, depending on the algorithm. While implementation details might also contribute to an increase in training time, it is not surprising given the architecture of Dreamer: before each environment interaction phase, Dreamer does many parameter optimization steps using already collected data. Thus, while being sample efficient, Dreamer needs much more CPU time compared to other approaches.

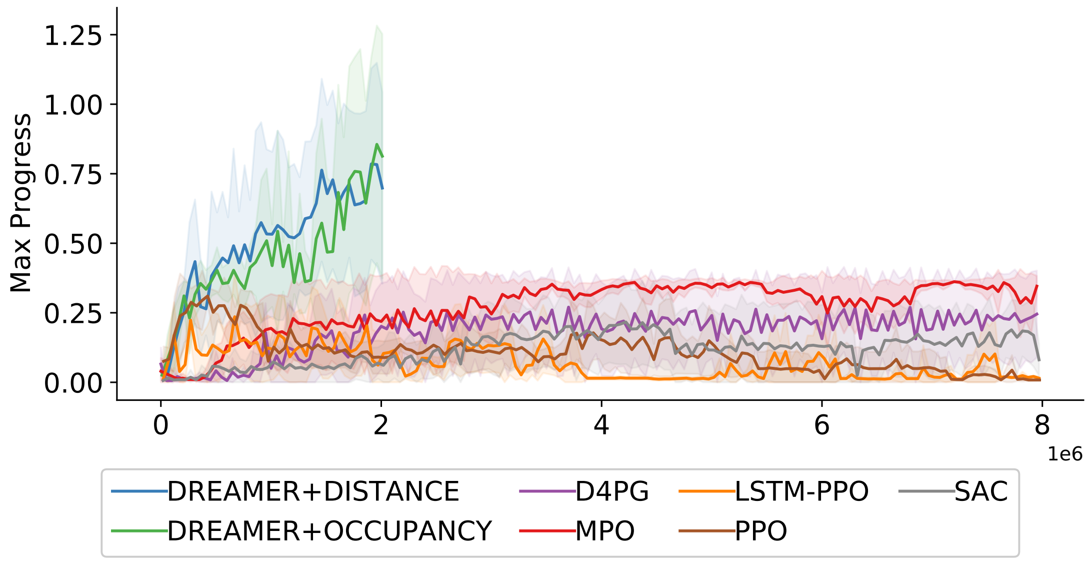
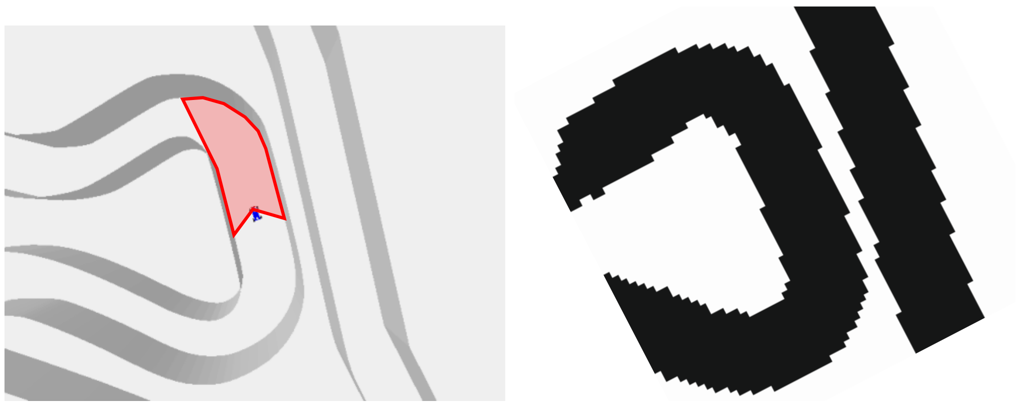


Figure 6.3: MAP for all agents on Austria during training, evaluated every 2×10^4 timesteps.

6.1.3 Reconstruction Loss Comparison

We trained two versions of Dreamer using different reconstruction losses, as described in chapter 4. Figures 6.1 and 6.3 already showed that reconstructing a local grid map from the latent state can improve the performance. As we discuss in section 6.2, we also observed that using the map reconstruction loss might lead to overfitting to the training track. Additionally, we also experimented with varying training horizons in imagination and compared the performance of Dreamer using these two reconstruction losses.
Figure 6.5 shows the comparison of different training horizons for the grid map reconstruction loss and the distance reconstruction loss on Austria. We trained Dreamer with different imagination horizons and observed, that using the grid map reconstruction loss scales better with increasing horizons than distance reconstruction. One explanation for this seems to be the global knowledge about a map that is encoded into the dynamics model. Predictions of a dynamics model, that is trained to reconstruct LiDAR scans, are accurate for a short period into the future because the model does not know what comes behind a corner, for example. This is visualized in fig. 6.4a, where the LiDAR scan covers the area just before a corner. Because the observation might look the same in many spots on the track, the model is not able to make accurate predictions about states far in the future. Dynamics models trained on maps, however, might be able to remember tracks better and are more accurate when predicting long horizons. In the example scene shown in fig. 6.4a, a model having knowledge about the map can predict *around the corner*, as shown in fig. 6.4b.



(a) Top view on race car with visualized LiDAR scan.

(b) Grid map reconstruction.

Figure 6.4: Map reconstruction.

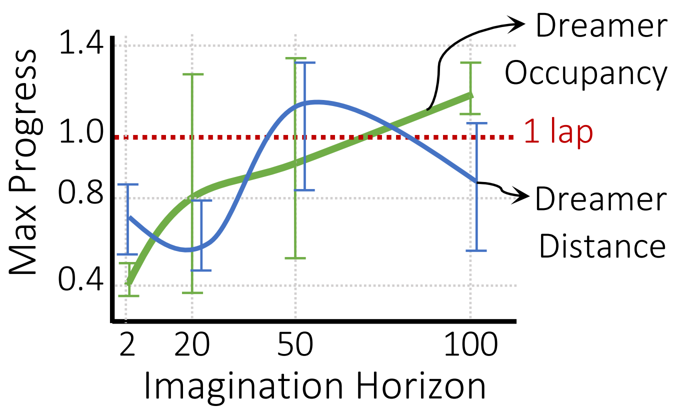


Figure 6.5: Peformance of Dreamer on Austria using a range of different training horizons. The green line shows the average performance over 5 runs of Dreamer using grid map reconstruction loss, the blue line the performance using LiDAR observation reconstruction loss. The vertical bars show the variance in performance.

In our experiments, we focused on the sensitivity of Dreamer to variations of the parameters *Imagination Horizon* and *Action Repeat*. We carried out our experiments by fixing one configuration and systematically evaluate variations of the other parameters. Interestingly, the performance of Dreamer using distance reconstruction drops for horizons greater than 2 and starts increasing again for horizons ranging from 20 to 50. Unfortunately, we can not explain this initial drop in performance yet and refer to future work where we want to better explore the training dynamics of Dreamer in the latent state space.

6.2 Generalization to Holdout Tracks

In this experiment, we aim to evaluate the ability of RL algorithms to generalize learned driving skills. In order to investigate if agents overfit to training tracks, we let them drive on tracks that are different from the track on which they were trained on. We compared Dreamer with two different reconstruction models to MPO and FTG, a vehicle control algorithm that is often used in racing competitions [SG12]. The choice of MPO was motivated by the observation, that this algorithm was the most competitive model-free algorithm compared to other baselines. Additionally, we included the results of FTG, as agents trained with MPO often fail to complete a whole lap and FTG is a competitive baseline for racing[1].

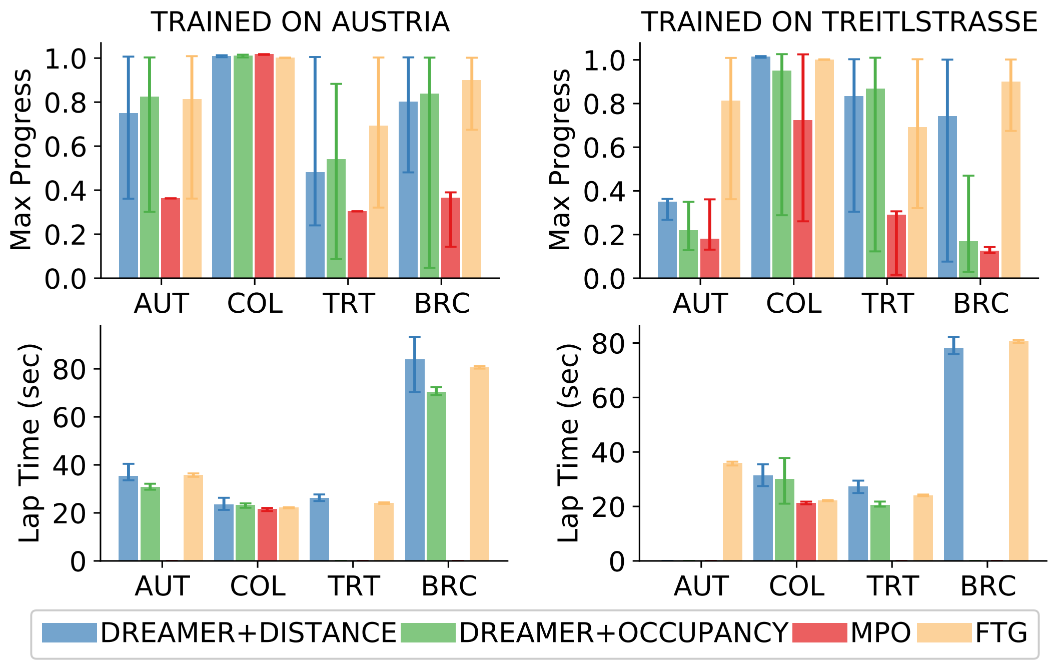


Figure 6.6: **Top:** Average MAP over 10 episodes for agents trained on Austria and Treitlstrasse, respectively. Whiskers show the minimum and maximum values. **Bottom:** Average lap time for agents that completed a full lap on a given track.

In fig. 6.6, we show the MAP reached within the evaluation time window as well as the lap time if an algorithm was able to complete a full lap. For this experiment, we trained Dreamer and MPO on Austria and Treitlstrasse, respectively, and subsequently evaluated them on all tracks. The top row in fig. 6.6 shows the average MAP over 10 episodes visualized as bar-charts with whiskers from minimum to maximum MAP. Similarly, the bottom row shows the average lap time of the agents which were able to complete a lap, with minimum and maximum values indicated by whiskers.

[1]Experiences and thoughts from the 3rd F1/10th competition

The results show, that MPO could not complete a full lap on Austria, Treitlstrasse, and Barcelona, regardless of the training track. Therefore, there are no lap times reported for these tracks for MPO. However, MPO showed competitive performance and lap times on Columbia when being trained on Austria or Treitlstrasse. This is interesting because it was not able to complete a full lap on these tracks. Confirming the results of previous experiments, both versions of Dreamer were able to complete a full lap on any track they were trained on. After training Dreamer-Distance on Austria, this agent was able to complete at least one lap on all other tracks with lap-times comparable to agents using FTG. We observed, that Dreamer-Occupancy was able to complete all tracks except Treitlstrasse, after being trained on Austria. Furthermore, this agent showed slightly better results on the time-trial task than Dreamer-Distance and FTG. On the contrary, after being trained on Treitlstrasse, both agents did perform worse on the holdout tracks compared to agents being trained on Austria. Noteworthy, no Dreamer agent was able to complete a full lap on Austria when being trained on Treitlstrasse, and Dreamer-Occupancy only completed full laps on the training track and Columbia.
From the observations of this experiment, we conclude that Dreamer agents that are using LiDAR reconstruction are more robust to variations of the tracks than agents which are trained using occupancy map reconstruction. This indicates, that Dreamer-Occupancy might overfit the maps that it is trained on and therefore does not generalize its driving behavior to tracks that are significantly different. This explanation is also supported by the observation, that Dreamer-Occupancy drives more aggressively than Dreamer-Distance and thus achieves better lap times on training tracks or tracks that are sufficiently similar to them (e.g. Austria and Barcelona), but fails to complete a full lap on other tracks. Another observation is, that if the training track contains enough variability, such as curves of variable difficulty, agents are able to learn more general driving behavior. Presumably, this also explains the performance difference of agents that are trained on Austria and agents that are trained on Treitlstrasse, as Austria has more variability of curves than Treitlstrasse.

6.3 Real-World Performance Evaluation

Training agents in simulation before deploying them to real robots is the default workflow when applying RL in robotics. While this yields many advantages and, in most cases, is inevitable, the transition from simulation to the real platform is often problematic, because of the divergence of simulators and real-world robots. To evaluate the driving behavior of Dreamer on real-world cars, we deployed an agent, that was trained in a simulated version of Treitlstrasse to our hardware platform and set up three scenarios on our test track. These experiments were conducted on our real-world test track in a lecture hall at the University of Technology in Vienna. A video of all three real-world experiments is available on the YouTube-Channel of the Cyber-Physical-Systems research group[2]. Figure 6.7 shows footage from the experiments on the test track. The two images

[2] `https://www.youtube.com/watch?v=IlN3vJxC30w`

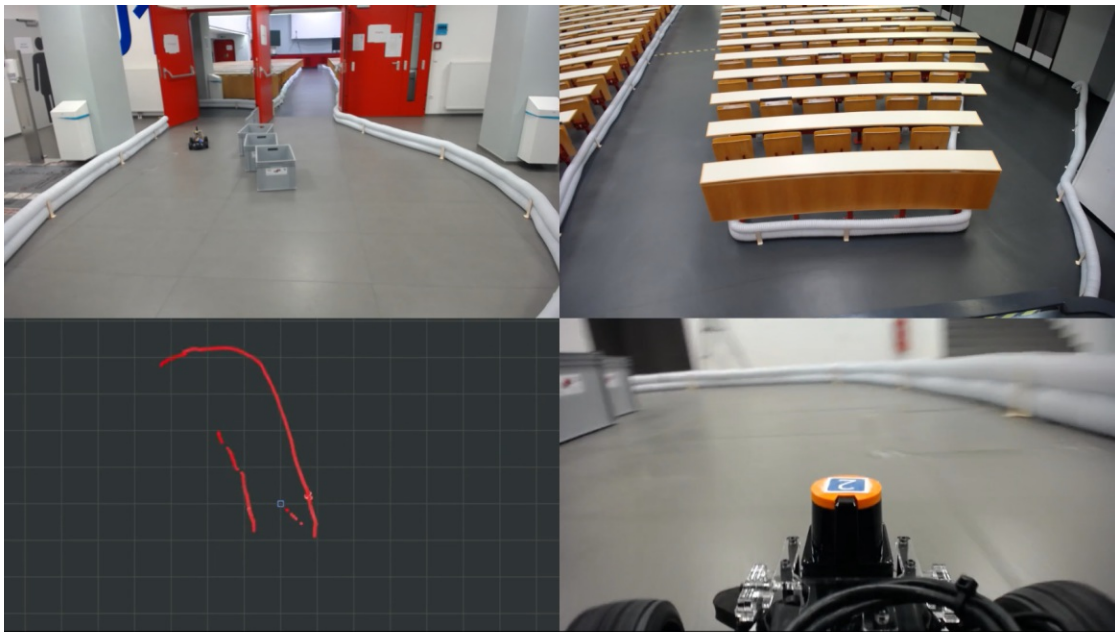


Figure 6.7: Footage of the real-world test setup in Treitlstrasse. From top left, clockwise: camera in turn 5, camera in turn 1 and 2, on-board camera, LiDAR scan.

in the top row are captured by webcams located on both ends of the track (turns 5 and 1, see fig. 5.4a). The bottom row shows onboard camera footage and LiDAR observations of the car while driving through turn 5. For each experiment outlined below, we recorded approximate lap times, presented in table 6.1.

Table 6.1: Lap times of real-world experiments.

Experiment	**Lap Time**
Driving a Full Lap	$\sim 28s$
Driving in Reverse Direction	$\sim 31s$
Unexpected Obstacles	$\sim 32s$

Driving a full lap. The first experiment was to evaluate if an agent, that is able to complete a full lap in simulation, can transfer the learned behavior successfully to the real car. For this experiment, we tested both types of agents trained with Dreamer, Dreamer-Distance, and Dreamer-Occupancy. While Dreamer-Distance was able to complete a full lap without collisions, we observed that Dreamer-Occupancy learned an aggressive driving behavior, leading to frequent crashes throughout the course. We believe, that this could be partially caused by small differences between the simulated tracks and the real-world track, as Dreamer-Occupancy trains its dynamics model by reconstructing parts of the track. The differences are introduced by moving the track-boundaries when crashing into them during testing or by frequent reassembly of the track.

Driving in reverse direction. After confirming that agents, that were trained in simulation, are able to drive safely on the real-world version of the training track, we investigated the behavior of Dreamer-Distance when driving in the reversed direction on the same track. By driving in the opposite direction, we aim to provide a *new* real-world track, as we did not have the space to build another track. The agent was also able to complete a full lap in this experiment. However, we observed that the agent swerves from left to right more heavily compared to the previous experiment. Furthermore, the lap time was around 3 seconds higher, compared to the agent driving in the default direction, as shown in table 6.1.

Unexpected Obstacles. In a final experiment, we tested the behavior of the agent when unexpected obstacles appear on the track. Although such a scenario was never covered in the training data, the agent was able to navigate around the obstacles safely. Figure 6.8 shows the obstacles from the agent's point of view, while the car navigates around them.

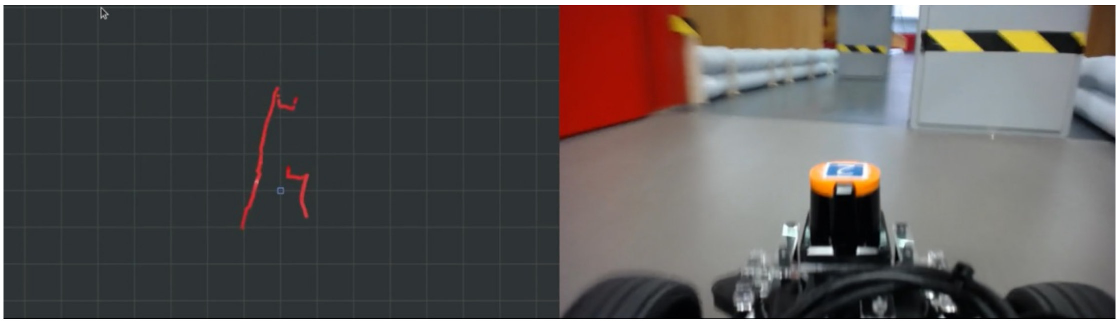


Figure 6.8: Obstacles captured by LiDAR and camera on the vehicle.

CHAPTER 7

Conclusion

In this chapter, we summarize the core findings of this thesis and discuss the experimental results. Section 7.1 gives answers to the research questions we formulated in the beginning and links them to the results we obtained in the course of the experiments. Furthermore, in section 7.2, we give a brief outlook on our future research plans.

7.1 Research Questions

In the following we relate the results and insights that we obtained throughout this work to the research questions defined in chapter 1.

How do world model RL algorithms compare to the state of the art model-free RL algorithms with respect to sample efficiency and performance?

We conducted extensive experiments to compare the performance of Dreamer, a world model algorithm, to various, state-of-the-art model-free RL algorithms. The metric we designed to compare the results of different algorithms was the MAP that was reached over a fixed time window of 40 seconds. According to the learning results that we observed, Dreamer achieves superior performance, while being much more sample efficient than the baseline algorithms. Furthermore, we observed that Dreamer was also able to achieve lap times comparable to an FTG agent, a vehicle control algorithm that was used by several winning teams of the F1TENTH competitions. It is noteworthy, that the model-free algorithms did not achieve to complete a full lap on most of the tracks.
We expected that Dreamer was more sample efficient than model-free approaches because its latent dynamics model allows training a policy in imagination instead of using sampled experience. Also, the performance values reported in the original paper suggest that world model algorithms do exceed the performance of model-free baselines, at least given this amount of training data. However, we acknowledge that the difference in performance could be, at least in parts, due to other reasons than algorithmic superiority. For instance,

it might require more hyperparameter tuning or better model architectures. However, we utilized our computational resources to their limits to provide fair results. Thus, we can state that world model algorithms show better performance, given this amount of training data and that more computational resources would allow testing this hypothesis in higher sample regimes.

Are world models able to adapt to unknown situations at test time?

The results of the track generalization experiments suggest that agents that were trained with Dreamer on one track can adapt well to different tracks at test time. We observed that the performance of cross-track generalization depends on the choice of the reconstruction model. While the map reconstruction approach led to better performance on the training track, the observation reconstruction model allowed the agent to better adapt to other tracks. Another form of adaption to unknown situations could be observed during real-world experiments. Dreamer was able to complete several laps on the real vehicle, even though the agent was trained in simulation without any domain randomization techniques. Furthermore, our real-world experiments showed that Dreamer was also able to navigate the track in the opposite direction and even after we placed obstacles on it, which were not present during training.
The experiments suggest that Dreamer is at least able to adapt to changes of the tracks, such as course layout modifications and much higher slippage compared to simulation. However, more adaption experiments, such as changing properties of the vehicle (e.g. weight), would provide insight into the adaption capabilities of Dreamer to changes in the vehicle dynamics.

Do world models, that are learned in simulation, facilitate safe real-world navigation for autonomous vehicles?

Our experiments on the real-world track showed that Dreamer agents, that were trained in simulation, were able to navigate several laps on the real-world track without crashing into the track boundaries. We observed, that the choice of reconstruction model also influences the safety behavior of the agent. The real-world experiments showed that agents that were trained with the map reconstruction model had a more aggressive driving style compared to the observation reconstruction approach, which occasionally led to crashes. We believe that this is related to overfitting to the track and, as a consequence, exaggerated confidence about the future states. In future experiments, we aim to investigate how well world model algorithms can avoid crashes when dynamic obstacles, such as other vehicles, are present.

7.2 Outlook and Future Work

Throughout this work, we identified several future research directions which we want to explore in more detail. We are planning to work on some problems that we observed with our current approach, in the near future. For instance, while the agent was able to drive without crashes, we observed that the controls showed heavy oscillations. This

could potentially damage actuators and is in general not desired. Our current work focus is to resolve this issue by implementing approaches we found in the continuous control literature.
After optimizing our current approach, we aim to explore and transfer our knowledge to the field of competitive multi-agent RL in the context of autonomous racing. The following list gives a brief overview of the questions that we want to answer:

- How can we trace back the contribution of a single-agent decision to the cumulative success of its team?
- How can we learn a predictive model of the opponent's behavior from just a few seconds of interaction, to estimate their future actions and action responses?
- How can we formulate team-based multi-agent racing competitions as hierarchical RL problems?
- How can we combine deep RL with game-theoretic approaches to obtain better strategies for autonomous racing scenarios?

By answering these questions, we hope to contribute to the state of the art of RL and multi-agent systems in general.

Appendix

A.1 Hyperparameters

Parameter	Value	Search Space
Initial Dataset Size	5000 steps	-
Update Steps	100	-
Batch Size	50	-
Sequence Length	50	-
Action Repeat	8	4, 8
Imagination Horizon	15	2, 20, 50, 100
Learning Rate θ	6e-4	-
Learning Rate ψ	8e-5	-
Learning Rate ϕ	8e-5	-
Discount	0.99	-
λ	0.95	-

Table A.1: Hyperparameters for Dreamer.

Parameter	Value	Search Space	Study Result
Policy Learning Rate	10^{-4}	$[10^{-5}, 10^{-3}]$	0.000672035
Critic Learning Rate	10^{-4}	$[10^{-5}, 10^{-3}]$	0.000903657
Batch Size	256	-	-
Epsilon	0.1	-	-
Epsilon Mean	10^{-3}	$[5 \times 10^{-4}, 10^{-3}]$	0.000654296
Epsilon Stddev.	10^{-6}	$[5 \times 10^{-7}, 10^{-5}]$	8×10^{-7}
Replay Buffer Size	5×10^{5}	-	-

Table A.2: Hyperparameters and study results for MPO.

Parameter	Value	Search Space	Study Result
Policy Learning Rate	10^{-4}	$[10^{-5}, 10^{-3}]$	0.00071
Critic Learning Rate	10^{-4}	$[10^{-5}, 10^{-3}]$	0.00001
Batch Size	256	-	-
Sigma	0.3	-	-
Atoms	51	-	-
Discount	0.99	-	-
Replay Buffer Size	5×10^{5}	-	-

Table A.3: Hyperparameters and study results for D4PG.

Parameter	Value	Search Space	Study Result
Learning Rate	3×10^{-4}	$[10^{-5}, 3 \times 10^{-3}]$	0.00131
Batch Size	64	-	-
Discount	0.99	-	-
GAE-λ	0.95	-	-
Clip Range	0.2	$[0.1, 0.3]$	0.3
Entropy Coef.	0.0	$[0.0, 0.01]$	0.001
Value Function Coef.	0.5	$[0.5, 1.0]$	0.8

Table A.4: Hyperparameters and study results for PPO.

Parameter	Value	Search Space	Study Result
Learning Rate	3×10^{-4}	-	-
Batch Size	256	-	-
Tau	0.005	-	-
Discount	0.99	-	-
Replay Buffer Size	5×10^{5}	-	-

Table A.5: Hyperparameters and study results for SAC.

List of Figures

List of Tables

List of Algorithms

Acronyms

A3C	Asynchronous Advantage Actor-Critic. 21
CNN	Convolutional Neural Network. 21, 25
D4PG	Distributed Distributional DDPG. 15, 41, 43, 44, 58, 61
DDPG	Deep Deterministic Policy Gradient. 14–16
ELBO	evidence lower bound. 16, 24
EM	Expectation Maximization. 16, 17
ESC	Electronic Speed Control. 35
FTG	Follow the Gap. 43, 48, 49, 53
KL	Kullback-Leibler. 14
LiDAR	Light Detection and Ranging. 7, 21, 25–27, 35, 46, 47, 49–51, 59, 60
LSTM	Long Short-Term Memory. 23, 42–44
MAP	Maximum Absolute Progress. 39, 42–44, 46, 48, 53, 59, 60
MDP	Markov Decision Process. 3, 5–9, 23
ML	Machine Learning. 1, 5
MLP	Multi-Layer Perceptron. 25, 26, 40–42
MPC	Model Predictive Control. 21
MPO	Maximum a Posteriori Policy Optimization. 16, 17, 20, 41, 43, 44, 48, 49, 58, 61
MSE	Mean Squared Error. 13

POMDP Partially Observable MDP. 5, 7, 8

PPO Proximal Policy Optimization. 14, 20, 41–44, 58, 61

RL Reinforcement Learning. 1–3, 5, 8–11, 13, 15–17, 19–21, 31, 32, 35, 45, 48, 49, 53, 55, 59

ROS Robot Operating System. 35, 36

RSSM Recurrent State Space Model. 23, 24, 40, 61

SAC Soft Actor-Critic. 15, 16, 20, 21, 41, 43, 58, 61

SLAM Self-Localization and Mapping. 36

TRPO Trust-Region Policy Optimization. 14, 20

Bibliography

[AOS+16] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete Problems in AI Safety, 2016.

[AST+18] Abbas Abdolmaleki, Jost Tobias Springenberg, Yuval Tassa, Remi Munos, Nicolas Heess, and Martin Riedmiller. Maximum a Posteriori Policy Optimisation. In *International Conference on Learning Representations*, 2018.

[ASY+19] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, KDD '19, page 2623–2631, New York, NY, USA, 2019. Association for Computing Machinery.

[BB20] Varundev Suresh Babu and Madhur Behl. f1tenth. dev-An Open-source ROS based F1/10 Autonomous Racing Simulator. In *16th Int. Conf. on Automation Science and Engineering (CASE)*, pages 1614–1620. IEEE, 2020.

[BBBK11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.

[BCMS08] F. Braghin, F. Cheli, S. Melzi, and E. Sabbioni. Race driver model. *Computers & Structures*, 86(13):1503–1516, 2008. Structural Optimization.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016.

[Bel03] Richard Ernest Bellman. *Dynamic Programming.* Dover Publications, Inc., USA, 2003.

[Ber07] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. II.* Athena Scientific, 3rd edition, 2007.

[BMHB$^+$18] Gabriel Barth-Maron, Matthew W. Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva TB, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributional Policy Gradients. In *International Conference on Learning Representations*, 2018.

[BRW$^+$19] Matthew Botvinick, Sam Ritter, Jane X. Wang, Zeb Kurth-Nelson, Charles Blundell, and Demis Hassabis. Reinforcement Learning, Fast and Slow. *Trends in Cognitive Sciences*, 23(5):408–422, 2019.

[BYC13] J. Bergstra, D. Yamins, and D. D. Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page I–115–I–123. JMLR.org, 2013.

[CB19] Erwin Coumans and Yunfei Bai. PyBullet, a Python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, 2016–2019.

[CvMG$^+$14] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Fethi Bougares, Holger Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. 06 2014.

[CWHL19] Fabian Christ, Alexander Wischnewski, Alexander Heilmeier, and B. Lohmann. Time-optimal trajectory planning for a race car considering variable tyre-road friction coefficients. *Vehicle System Dynamics*, 59:588 – 612, 2019.

[DDVG19] Ankit Dhall, Dengxin Dai, and Luc Van Gool. Real-time 3D Traffic Cone Detection for Autonomous Driving. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 494–501, 2019.

[DR11] Marc Peter Deisenroth and Carl Edward Rasmussen. PILCO: A Model-Based and Data-Efficient Approach to Policy Search. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, page 465–472, Madison, WI, USA, 2011. Omnipress.

[EIS$^+$20] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation Matters in Deep RL: A Case Study on PPO and TRPO. In *International Conference on Learning Representations*, 2020.

[FSK$^+$21] Florian Fuchs, Yunlong Song, Elia Kaufmann, Davide Scaramuzza, and Peter Dürr. Super-Human Performance in Gran Turismo Sport Using Deep Reinforcement Learning. *IEEE Robotics and Automation Letters*, 6(3):4257–4264, 2021.

[FvHM18] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1587–1596. PMLR, 10–15 Jul 2018.

[Gor95] Geoffrey J. Gordon. Stable Function Approximation in Dynamic Programming. In Armand Prieditis and Stuart Russell, editors, *Machine Learning Proceedings 1995*, pages 261–268. Morgan Kaufmann, San Francisco (CA), 1995.

[Haf19] Danijar Hafner. Dreamer Codebase. `https://github.com/danijar/dreamer`, 2019.

[HLBN20] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to Control: Learning Behaviors by Latent Imagination. In *International Conference on Learning Representations*, 2020.

[HLF+19] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning Latent Dynamics for Planning from Pixels. In *International Conference on Machine Learning*, pages 2555–2565, 2019.

[Hok20] Hokuyo Automatic USA Corporation. *Scanning Laser RangeFinder Smart-URG mini UST-10LX Specification*, 12 2020.

[HPZ+18] T. Haarnoja, Vitchyr H. Pong, Aurick Zhou, Murtaza Dalal, P. Abbeel, and Sergey Levine. Composable Deep Reinforcement Learning for Robotic Manipulation. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6244–6251, 2018.

[HRE+18] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable Baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HS18] David Ha and Jürgen Schmidhuber. Recurrent World Models Facilitate Policy Evolution. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[HSA+20] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A Research Framework for Distributed Reinforcement Learning. *arXiv preprint arXiv:2006.00979*, 2020.

[HWMZ20] Lukas Hewing, Kim P. Wabersich, Marcel Menner, and Melanie N. Zeilinger. Learning-Based Model Predictive Control: Toward Safe Learning in Control. *Annual Review of Control, Robotics, and Autonomous Systems*, 3(1):269–296, 2020.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870. PMLR, 10–15 Jul 2018.

[HZH$^+$19] T. Haarnoja, Aurick Zhou, Sehoon Ha, J. Tan, G. Tucker, and Sergey Levine. Learning to Walk via Deep Reinforcement Learning. *ArXiv*, abs/1812.11103, 2019.

[JdT$^+$18] M. Jaritz, R. de Charette, M. Toromanoff, E. Perot, and F. Nashashibi. End-to-End Race Driving with Deep Reinforcement Learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2070–2075, 2018.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2015.

[KBP13] Jens Kober, J. Bagnell, and Jan Peters. Reinforcement Learning in Robotics: A Survey. *The International Journal of Robotics Research*, 32:1238–1274, 09 2013.

[KHLZ19] Juraj Kabzan, Lukas Hewing, Alexander Liniger, and Melanie N. Zeilinger. Learning-Based Model Predictive Control for Autonomous Racing. *IEEE Robotics and Automation Letters*, 4(4):3363–3370, 2019.

[KKFH07] Jonathan Ko, D. Klein, D. Fox, and D. Hähnel. Gaussian Processes and Reinforcement Learning for Identification and Control of an Autonomous Blimp. *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 742–747, 2007.

[KLC98] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99–134, 1998.

[KVR+20] Juraj Kabzan, M. I. Valls, Victor Reijgwart, Hubertus Franciscus Cornelis Hendrikx, Claas Ehmke, Manish Prajapat, A. Bühler, N. Gosala, Mehak Gupta, R. Sivanesan, Ankit Dhall, E. Chisari, Napat Karnchanachari, Sonja Brits, Manuel Dangel, Inkyu Sa, Renaud Dubé, Abel Gawel, Mark Pfeiffer, Alexander Liniger, J. Lygeros, and R. Siegwart. AMZ Driverless: The Full Autonomous Racing System. *ArXiv*, abs/1905.05150, 2020.

[KW14] Diederik P. Kingma and M. Welling. Auto-Encoding Variational Bayes. *CoRR*, abs/1312.6114, 2014.

[LA14] Sergey Levine and Pieter Abbeel. Learning Neural Network Policies with Guided Policy Search under Unknown Dynamics. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[LB14] T. Lipp and Stephen P. Boyd. Minimum-time speed optimisation over a fixed path. *International Journal of Control*, 87:1297 – 1311, 2014.

[LFDA16] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-End Training of Deep Visuomotor Policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[LHP+15] Timothy Lillicrap, Jonathan Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, 09 2015.

[LJD+18] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018.

[LL01] F. Lamiraux and J. . Lammond. Smooth motion planning for car-like vehicles. *IEEE Transactions on Robotics and Automation*, 17(4):498–501, 2001.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.

[MKV+18] A. Mahmood, Dmytro Korenkevych, Gautham Vasan, W. Ma, and J. Bergstra. Benchmarking Reinforcement Learning Algorithms on Real-World Robots. In *CoRL*, 2018.

[MM02] Remi Munos and Andrew Moore. Variable Resolution Discretization in Optimal Control. *Machine Learning*, 49:291–, 11 2002.

[Ope21] OpenAI. OpenAI - Spinning Up. `https://spinningup.openai.com/en/latest/index.html`, 2021. Online; accessed 02 March 2021.

[OSA+19] Matthew O'Kelly, Varundev Sukhil, Houssam Abbas, Jack Harkins, Chris Kao, Yash Vardhan Pant, Rahul Mangharam, Dipshil Agarwal, Madhur Behl, Paolo Burgio, and Marko Bertogna. F1/10: An Open-Source Autonomous Cyber-Physical Platform, 2019.

[PAZA18] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-Real Transfer of Robotic Control with Dynamics Randomization. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018.

[QCG+09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. volume 3, 01 2009.

[RB20] Ugo Rosolia and Francesco Borrelli. Learning How to Autonomously Race a Car: A Predictive Control Approach. *IEEE Transactions on Control Systems Technology*, 28(6):2713–2719, 2020.

[RGLR16] Aravind Rajeswaran, Sarvjeet Ghotra, Sergey Levine, and Balaraman Ravindran. EPOpt: Learning Robust Neural Network Policies Using Model Ensembles. *CoRR*, abs/1610.01283, 2016.

[RHE+19] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable Baselines3. `https://github.com/DLR-RM/stable-baselines3`, 2019.

[RK03] Carl Edward Rasmussen and Malte Kuss. Gaussian Processes in Reinforcement Learning. In *NIPS*, pages 751–758, 2003.

[SAS+20] H. Francis Song, Abbas Abdolmaleki, Jost Tobias Springenberg, Aidan Clark, Hubert Soyer, Jack W. Rae, Seb Noury, Arun Ahuja, Siqi Liu, Dhruva Tirumala, Nicolas Heess, Dan Belov, Martin Riedmiller, and Matthew M. Botvinick. V-MPO: On-Policy Maximum a Posteriori Policy Optimization for Discrete and Continuous Control. In *International Conference on Learning Representations*, 2020.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[Ser21] Sergey Levine. CS285 - Deep Reinforcement Learning. `http://rail.eecs.berkeley.edu/deeprlcourse/`, 2021. Online; accessed 02 March 2021.

[SG12] Volkan Sezer and Metin Gokasan. A novel obstacle avoidance algorithm: “Follow the Gap Method”. *Robotics and Autonomous Systems*, 60(9):1123–1134, 2012.

[SHC+17] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning, 2017.

[SHM+16] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[SL17] Fereshteh Sadeghi and Sergey Levine. CAD2RL: Real Single-Image Flight Without a Single Real Image. In *Proceedings of Robotics: Science and Systems*, Cambridge, Massachusetts, July 2017.

[SLA+15] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust Region Policy Optimization. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897. PMLR, 07–09 Jul 2015.

[SML+16] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[Sut91] Richard S. Sutton. Dyna, an Integrated Architecture for Learning, Planning, and Reacting. *SIGART Bull.*, 2(4):160–163, July 1991.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, 2017.

[TBF05] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[TDK17] TDK Corporation. *World’s Lowest Power 9-Axis MEMS MotionTracking™ Device*, 2 2017.

[TDM+18] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy Lillicrap, and Martin Riedmiller. DeepMind Control Suite. 01 2018.

[TFR+17] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter 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)*, pages 23–30, 2017.

[TL05] E. Todorov and Weiwei Li. A generalized iterative LQG method for locally-optimal feedback control of constrained nonlinear stochastic systems. In *Proceedings of the 2005, American Control Conference, 2005.*, pages 300–306 vol. 1, 2005.

[TR96] John Tsitsiklis and Benjamin Roy. Featured-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94, 03 1996.

[VBC+19] Oriol Vinyals, Igor Babuschkin, Wojciech Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John Agapiou, Max Jaderberg, and David Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575, 11 2019.

[VBL+20] José L. Vázquez, Marius Brühlmeier, Alexander Liniger, Alisa Rupenyan, and John Lygeros. Optimization-Based Hierarchical Motion Planning for Autonomous Racing. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2397–2403, 2020.

[vH12] Hado van Hasselt. *Reinforcement Learning in Continuous State and Action Spaces*, pages 207–251. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[VHR+18] Miguel I. Valls, Hubertus F.C. Hendrikx, Victor J.F. Reijgwart, Fabio V. Meier, Inkyu Sa, Renaud Dubé, Abel Gawel, Mathias Bürki, and Roland Siegwart. Design of an Autonomous Racecar: Perception, State Estimation and System Integration. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2048–2055, 2018.

[WBC+19] Tingwu Wang, Xuchan Bao, Ignasi Clavera, Jerrick Hoang, Yeming Wen, Eric Langlois, Shunshi Zhang, Guodong Zhang, Pieter Abbeel, and Jimmy Ba. Benchmarking Model-Based Reinforcement Learning, 2019.

[WML+17] Yuhuai Wu, Elman Mansimov, Shun Liao, Roger Grosse, and Jimmy Ba. Scalable Trust-Region Method for Deep Reinforcement Learning Using Kronecker-Factored Approximation. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 5285–5294, Red Hook, NY, USA, 2017. Curran Associates Inc.

[WSBR15] Manuel Watter, Jost Tobias Springenberg, Joschka Boedecker, and Martin Riedmiller. Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images. In *Proceedings of the 28th International*

Conference on Neural Information Processing Systems - Volume 2, NIPS'15, page 2746–2754, Cambridge, MA, USA, 2015. MIT Press.

[ZQW20] W. Zhao, J. P. Queralta, and T. Westerlund. Sim-to-Real Transfer in Deep RL for Robotics: a Survey. In *IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 737–744, 2020.

[ZVS$^+$19] Marvin Zhang, S. Vikram, Laura Smith, P. Abbeel, M. Johnson, and Sergey Levine. SOLAR: Deep Structured Representations for Model-Based Reinforcement Learning. In *ICML*, 2019.

[Ås65] K.J Åström. Optimal control of Markov processes with incomplete state information. *Journal of Mathematical Analysis and Applications*, 10(1):174–205, 1965.