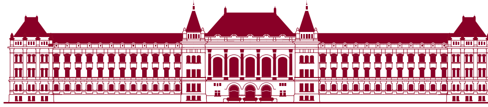


MSc THESIS TOPIC

Péter Béla Almási
Graduating Student in Informatics

Real-world Deep Reinforcement Learning via Simulation for Autonomous Driving

Deep Reinforcement Learning (DRL) has been successfully used to solve different challenges, e.g. complex board and computer games, recently. However, solving real-world robotics tasks with DRL seems to be a more difficult challenge. The desired approach would be to train the agent in a simulator and transfer it to the real world. Even training agents in an autonomous driving simulator is a challenging task, because most DRL methods still lack mathematical fundamentals and are unstable. Furthermore, models trained in a simulator tend to have a decreased performance in real-world environments due to the differences.
The goal of the thesis is to investigate the possibilities and performance of deep reinforcement learning agents in the simulation and to elaborate novel methods to run the same or the fine-tuned models in a real-world environment with competitive performance.
The following subtasks should be elaborated:

- **Overview** the scientific papers about deep reinforcement learning (especially for autonomous driving) and simulation-to-real-world transformation of DRL agents.
- **Investigate** the possible simulation environments and choose one.
- **Design, build and train** at least two different deep reinforcement learning agents in the autonomous driving simulator.
- **Evaluate their performance** in the simulator and real-world environments in terms of training speed and accuracy.
- **Elaborate** novel methods to reduce the difference in performance between agents running in simulator and in real-world (e.g. by transfer learning, generalization techniques, domain randomization).
- **Evaluate the performance** of the proposed methods in both environments.
- **Investigate** the possibility of introducing the proposed method to other simulation and real-world environments.
- **Prepare detailed documentation** about the work, summarize the results and write the conclusion and possible future work.

Academic supervisor: Bálint Gyires-Tóth, PhD
Co-supervisor: Robert Moni

Budapest, 12th March 2020

Gábor Magyar, PhD
/Head of Department/

Dept. of Telecommunications and Media Informatics
Faculty of Electrical Engineering and Informatics
Budapest University of Technology and Economics

H-1117 Budapest, Magyar tudósok krt. 2.
Phone: (+36 1) 463 2448 Fax: (+36 1) 463 3107
E-mail: magyar@tmit.bme.hu http://www.tmit.bme.hu

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Real-world Deep Reinforcement Learning via Simulation for Autonomous Driving

Master's Thesis

Author
Péter Béla Almási

Advisors
Dr. Bálint Gyires-Tóth
Róbert Moni

May 22, 2021

Contents

Kivonat i

Abstract ii

1 Introduction 1

2 Background 4

2.1 Deep learning 4

2.1.1 Deep learning for computer vision 6

2.2 Reinforcement learning 7

2.2.1 Q-learning 8

2.2.2 Deep Q-Networks 9

2.2.3 DQN extensions 10

2.2.4 Policy gradient methods 11

2.2.5 Proximal Policy Optimization 11

2.2.6 Reinforcement learning for autonomous driving 12

2.3 Sim-to-real transfer 13

2.3.1 Domain adaptation 14

2.3.2 Domain randomization 16

2.3.3 System identification 21

2.4 Preliminaries 21

3 Proposed method 22

3.1 Environment and simulation-to-real transfer 22
3.2 Observation transformation 24
3.3 Training the agent 25
3.3.1 Reward function 25
3.4 Action space post-processing 26

4 Environment **27**

4.1 Autonomous driving environments 27
4.2 Duckietown 30
4.2.1 Duckietowns 31
4.2.2 Duckiebots 32
4.2.3 Duckietown simulator 33

5 Implementation **35**

5.1 Simulation 35
5.2 Observation transformation 36
5.3 Training the agents 36
5.3.1 Deep Q-Networks 37
5.3.2 Proximal Policy Optimization 38
5.3.3 Reward function 39
5.4 Action post-processing 40
5.5 Reference agent 40

6 Evaluation and results **41**

6.1 Collected rewards 41
6.2 Evaluation method 41
6.3 Testing results 43
6.3.1 Effects of action post-processing 45

7 Summary **48**

Acknowledgements **51**

Bibliography **52**

Appendix **58**

A.1 Randomization values 58

HALLGATÓI NYILATKOZAT

Alulírott *Almási Péter Béla*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2021. május 22.

Almási Péter Béla
hallgató

Kivonat

A mély neurális hálózatok kiemelkedő figyelemben részesültek az elmúlt években. Segítségükkel számtalan különféle alkalmazási területen sikerült minden korábbinál jobb eredményeket elérni: például képfelismerési, objektumdetekciós, beszédfelismerési és -generálási, természetes nyelvfeldolgozási vagy idősorelemzési feladatokban is kiemelkedőnek bizonyultak. Ezek a modellek sok esetben még az embernél is pontosabban oldják meg a számukra kijelölt feladatot.

A mély megerősítéses tanulás a gépi tanulási algoritmusok azon csoportját foglalja magába, amelyekben egy ágens neurális hálózatok használatával képes megtanulni egy környezetben egy bizonyos cél elérését a megfelelő akciók végrehajtásával. Ezzel a módszerrel vált lehetővé, hogy számítógépes algoritmusok legyőzzék a világbajnokokat különféle tábla- és számítógépes játékokban, például a Go-ban vagy a StarCraft II-ben.

Azonban a mély megerősítéses tanulás használata nagyobb kihívást jelent olyan feladatokban, amelyekben például valós robotokat vagy járműveket szeretnénk vezérelni. Ilyen feladatoknál jellemzően az ágenseket egy szimulátorban tanítják, majd a kész modellt "átültetik" a valós eszközre. A megerősítéses tanulás alkalmazása önvezető járművek esetében már szimulátorban is egy nehéz kihívás, hiszen ezek az algoritmusok jellemzően instabilak, és nem rendelkeznek kellően megalapozott matematikai háttérrel. Továbbá a szimulátorban tanított ágensekre jellemző, hogy a valós környezetben történő használatkor jelentősen romlik a teljesítményük.

Dolgozatomban egy olyan mély megerősítéses tanulás alapú eljárást dolgoztam ki, amellyel lehetséges önvezető ágenseket tanítani szimulátor segítségével, és ezeket sikeresen át lehet ültetni valós járművekre is. Az ágensek a valós környezetben, valódi járműveken futtatva is hasonló pontosságot nyújtanak - a valós környezetből vett tanító minták nélkül.

A dolgozatban két ágenst ismertetek, melyeket különböző algoritmusok segítségével tanítottam. A módszerek ismertetése után összehasonlítom és kiértékelem az ágensek teljesítményét mindkét környezetben. Az eredményeket egy demonstrációs videóban is bemutatom, amelyre a hivatkozás megtalálható a dolgozatban.

Abstract

Deep neural networks have received great attention in recent years. These models have been successfully used to reach *state-of-the-art* results in many application scenarios: for example, image recognition, object localization, speech recognition and synthesis, natural language processing, time series analysis, etc. These models can often solve their specific tasks more accurately than an average human does.

Deep Reinforcement Learning (DRL) is a field of machine learning which enables software agents using neural networks to reach their goal in a virtual environment by taking the best possible actions. This technique has been successfully used to beat world champions in different board and computer games, for example, Go or StartCraft II.

However, solving tasks involving real-world machines, e.g. robots or autonomous vehicles, with DRL seems to be a more difficult challenge. The desired approach would be to use a simulator to train the agent in a virtual environment and transfer it to the real world. Training agents in an autonomous driving simulator is already a challenging task, as most DRL methods still lack mathematical fundamentals and are unstable. Furthermore, models trained in a simulator tend to suffer from severe performance degradation when transferred to the real-world environment due to the differences.

In this work, I propose a novel method for training autonomous driving agents in a simulator and transferring them to real-world vehicles. As a result, the agents are able to drive autonomously in the real-world environment with a similar performance without further training on real-world data.

Two agents are presented in this thesis, which were trained using different DRL algorithms. After describing the method, I analyze and compare the performance of the agents in both environments. The results are also presented in a demonstration video, which is referred to in the document.

Chapter 1

Introduction

Artificial intelligence has undergone tremendous development in recent years. Its methods are used in countless application scenarios to help our everyday life. For example, personal assistants, translators, online stores' recommendation systems, or even mobile phones' cameras all use such algorithms to provide more valuable services.

Among its areas, deep learning has become a very actively researched area recently. Thanks to the recent advances in the development of GPUs (Graphical Processing Units), the computing power available via cloud services, and the (often public) availability of enormous databases, training deep neural networks has become possible and efficient. They have been successfully used to reach *state-of-the-art* results in several application scenarios: for example, image recognition [1] [2], speech recognition and synthesis [3], natural language processing [4], reinforcement learning [5], and robotics [6]. The performance of these networks is often comparable to or even better than the average human's accuracy.

Deep learning has an important role in the automotive industry. Using the recent image processing algorithms, it became possible to understand the camera images of the cars [7], which is an important milestone in self-driving car development. However, the problem of autonomous driving is far from being completely solved, thus it is under very active research nowadays.

Deep Reinforcement Learning (DRL) is an exciting area of deep learning, where instead of using huge labeled or unlabeled datasets, an agent interacts with a dynamic environment and tries to learn an optimal control policy by taking a sequence of actions. The agent receives feedback from the environment, which describes how *good* its actions have been. It tries is to reach its goal by finding the appropriate sequence

of actions among different circumstances. DRL algorithms have been successfully used recently to overcome human players in complex board games, such as Go [5], or computer games, such as StarCraft II [8].

Nonetheless, using DRL algorithms to solve real-world robotics control or autonomous driving tasks seems to be a more challenging scenario. The desired approach would be to train an agent in a simulator and then use it in the real world. Using a simulated environment has several advantages: it is much safer, it makes collecting an unlimited amount of labeled data possible, and is a lot cheaper than using real devices. However, when models trained in a simulator are transferred to the real world, they tend to have serious degradation in their performance due to the differences and simplifications between the simulator and the real world. Besides, while the simulator makes it possible to evaluate slower methods, algorithms used on real robots need to be designed carefully such that they can be run in real-time with very little latency. Sim2Real refers to the concept of transferring robotics skills learned in a simulator to real devices. Currently, there are no Sim2Real methods that generally work for various robotics applications, thus it is of great importance to examine the possibilities of existing techniques and develop new methods for autonomous driving.

This thesis discovers the challenges and examines the possibilities of performing lane following in a miniaturized urbanized autonomous driving environment, focusing on transferring agents from the simulator to the real vehicles.

The problem is formalized to fit a Markov Decision Process (MDP). In each time step, the vehicle's sensors provide raw multidimensional sensory data in the form of a D-dimensional tensor $S_t \in \mathbb{R}^{d_{0,0} \times d_{0,1} \times \cdots \times d_{0,D}}$. This input can be, for example, a camera image of shape $\mathbb{R}^{h \times w \times c}$, or a point cloud measured by a LIDAR sensor. The vehicle is controlled in a C-dimensional continuous control space, i.e. in each time step it is controlled by providing a control command $A_t \in \mathbb{R}^C$. With this formalization, the desired control function has to realize a mapping $f_c : \mathbb{R}^{d_{0,0} \times d_{0,1} \times \cdots \times d_{0,D}} \rightarrow \mathbb{R}^C$ which enables the vehicle for lane following. For training, an autonomous driving simulator is used. The performance of the method is evaluated both in the simulated and the real-world environment. Therefore, the method has to be able to cope with the differences between the two environments.

I develop a complete pipeline for training an agent for autonomous lane following. I propose a deep reinforcement learning-based algorithm to train a vehicle to perform lane following in an autonomous driving simulator, which provides a simplified

version of the real-world environment. I analyze and develop Sim2Real methods to make it possible to transfer the model trained in the simulator to the real robot. I design and compare two agents, which are trained with different DRL algorithms. My method is designed such that it can be run in real-time on a computer with limited hardware resources.

The method is implemented and tested in the Duckietown environment [9]. It is an educational and research platform where small three-wheeled vehicles can be used to perform autonomous driving tasks, for example, lane following, or collision avoidance. The environment is open, inexpensive, and highly flexible. The vehicles' only sensor is a monocular camera with a fish-eye lens.

The key challenges that are addressed in this problem are:

- Training two agents in an autonomous driving simulator to perform lane following using deep reinforcement learning
- Transferring the trained agents into the real world
- Running the method in real-time with limited hardware resources

The rest of this thesis is organized as follows. Chapter 2 introduces the theoretical background and the relevant methods of deep learning, reinforcement learning, and sim-to-real transfer. Chapter 3 introduces the proposed method. Chapter 4 gives an overview of autonomous driving environments. Chapter 5 describes how the proposed method was implemented and how the agents were trained in the Duckietown environment. The evaluation method along with the results is presented in chapter 6. Finally, conclusions are drawn in chapter 7.

Chapter 2

Background

In this section, I present the most important papers and results related to my work. I begin by giving a brief overview of deep learning and especially its aspect on image processing; then I introduce the reinforcement learning paradigm and present the main results of this field from recent years. Finally, the difficulties and the methods of transforming RL agents from the simulator to the real world are introduced.

2.1 Deep learning

Machine Learning (ML) is the concept of algorithms that can generate knowledge from data. For example, given a large dataset and target values for each data point, these algorithms can be trained to predict the target values based on the similarities of the elements of the dataset. Machine learning algorithms build an inner mathematical model based on the training data, and can automatically extract useful information from the data and make decisions or create predictions without explicitly being programmed to do so. If these algorithms are trained well and with an appropriate, diverse dataset, they can create correct predictions on previously unseen data points. Machine learning algorithms can be used with different kinds of datasets (e.g. images, sound recordings, or tabular data) and thus can be used in countless application scenarios. For example, they can identify objects on pictures, recognize and imitate human speech, recommend new movies or products from a webshop based on our interests, create financial forecasts, analyze chemical molecules, or even play computer games.

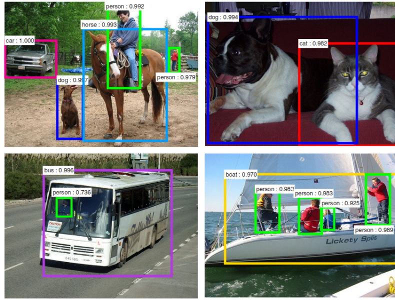
Machine learning techniques can be categorized into three different approaches. In the case of *supervised learning*, the algorithm is presented with both training data

and labels, and the goal is to create a function that predicts correct labels both for the training data and previously unseen examples. *Unsupervised learning* means that no target labels are available for the data; in this scenario, the algorithms have to find deeper structures in the data and, for example, identify anomalies or cluster it based on similarities. In the case of *reinforcement learning*, an agent interacts with a dynamic environment to learn an optimal control sequence to reach a specified goal. The agent can take one of some possible actions in each step and receives feedback (called reward) from the environment which describes how *good* its action was in the current state. The agent tries to learn an optimal policy that determines which action to take in each step to reach the goal. Reinforcement learning is widely used, for example, to train agents to play computer games or board games.

Deep learning refers to a family of machine learning algorithms that use *deep neural networks*. These networks consist of multiple layers, where each layer performs a mathematical operation on its input, which can be a matrix or a multidimensional tensor. Such operations can be, for example, matrix multiplication, activation (element-wise nonlinear function, e.g. sigmoid or ReLU [10]), 1- or 2-dimensional convolution, etc. The networks can be trained with error backpropagation to minimize the difference between its output and the target label.

In the cases of different applications and different kinds of datasets, the best results can be reached with specialized neural networks. For example, 2-dimensional Convolutional Neural Networks (CNNs) can be used for image recognition tasks; Recurrent Neural Networks (RNNs) and 1-dimensional CNNs can process text data, speech, or other kinds of time series; and feedforward neural networks can be used in other general applications, e.g. processing tabular data (feedforward layers are also widely used in CNNs or other types of networks).

The advantage of deep neural networks, which can be one of the reasons for their recent successes, is that their performance continues to improve as the amount of training data is increased. In contrast, other ML algorithms tend to stagnate after reaching a decent accuracy with a certain amount of data. The incredible amount of data in public or private datasets significantly contributed to the recent successes of neural networks. Besides, the developments of Graphical Processing Units (GPUs), lots of algorithmic improvements, and the computing power available via cloud services made training and using deep neural networks in various application scenarios possible and efficient.



(a) Object localization (source: [11])

(b) Semantic segmentation (source: [7])

Figure 2.1: Various computer vision tasks can be successfully solved by convolutional neural networks.

2.1.1 Deep learning for computer vision

Understanding the environment and recognizing the surrounding objects, e.g. other vehicles, pedestrians, and traffic signs is an important milestone in the development of self-driving vehicles. Cameras are widely used to perform this task, as they are cheaper and more accessible compared to LIDAR or RADAR sensors.

Deep neural networks have reached *state-of-the-art* results in several computer vision tasks. Convolutional neural networks [12] have been successfully applied to various image recognition and image processing tasks and outperformed former image processing methods. These networks are designed especially to successfully handle data with spatially correlated information and extract the relevant and required information from the input, e.g. pixels of images.

Image recognition, one of the most essential and most widely applied image processing tasks, consists of recognizing (one or more) objects in a given photo. This is widely used, for example, in smart camera applications or automatic image tagging algorithms. The ImageNet competition [13] is a large-scale competition focusing on categorizing images into 1000 categories based on the objects on them. This competition allows comparing the best image recognition algorithms. The fact that this competition is being won by convolutional neural networks since 2012 (e.g. [1]) proves that currently, CNNs are the *state-of-the-art* methods for image recognition tasks.

CNNs can be successfully used for more complex image processing tasks as well. Object localization and semantic segmentation are both useful in the automotive industry, as the cars not only have to determine what kind of objects (e.g. other cars, pedestrians, cyclists, etc.) are in the image but finding their exact location is

also an important part of the task. An example of these tasks is shown in Figure 2.1. In the case of object localization, in addition to recognizing the objects on the image, their locations need to be determined too. [11] shows an example of using CNNs for object localization. In the case of semantic segmentation, the image has to be partitioned into regions such that each region covers one object in the picture. It can also be viewed as a mapping from the pixels of the image to object categories, where each pixel has to be mapped to the object it is a part of. [7] shows an example of using CNNs for image segmentation.

2.2 Reinforcement learning

This section gives an overview of reinforcement learning and two approaches to solve reinforcement learning problems: Q-learning and policy gradient methods. The description and the formalization of this chapter took ideas from [14] and the contents of the university course *EL2805 Reinforcement Learning* taught by Alexandre Proutiere at KTH Royal Institute of Technology[1].

Reinforcement learning is an area of machine learning that can be modeled as an interaction between an intelligent agent and a dynamic environment. The agent tries to learn an optimal policy in order to reach a specified goal in the environment. In each step t, the agent, based on its current state s_t, selects one of the possible actions $a_t \in \mathcal{A}$. The environment, based on the agent's action a_t, computes the next state s_{t+1}, and also provides a scalar reward value r_t, which describes the quality of the selected action.

The goal of the agent is to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ which maps each state $s \in \mathcal{S}$ to an action $a \in \mathcal{A}$, to maximize the total discounted reward R earned by following a trajectory under π:

$$R = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]. \quad (2.1)$$

The discount factor, $\gamma \in (0, 1)$, is used to represent that future rewards are less important than present ones, and also for mathematical stability. In a general setting, both the transitions among the states and the policy π can be non-deterministic, thus the expectation of the total collected reward is calculated.

[1] `https://www.kth.se/student/kurser/kurs/EL2805?l=en` (Access date: 20 May 2021)

Some important concepts related to reinforcement learning are defined in the following.

Definition 1 (Model-based reinforcement learning). In the case of model-based reinforcement learning, the goal is to construct an internal model of the environment, which can accurately represent its transitions and outcomes. ▪

Definition 2 (Model-free reinforcement learning). In the case of model-free reinforcement learning, we use the experience of the agent to learn simpler quantities about the environment (e.g. state action values), from which we can derive a good policy. ▪

Definition 3 (On-policy learning). In the case of on-policy learning, the same policy is used for collecting experiences whose values are changed during training. ▪

Definition 4 (Off-policy learning). In the case of off-policy learning, we use a separate behavior policy π_b for collecting experiences during training, while we train our policy π. ▪

2.2.1 Q-learning

Reinforcement learning algorithms are applicable to problems that fit the Markov Decision Process (MDP). A common way to do so is called Q-learning. In this algorithm, we try to calculate or learn the values of the state action value function, also called Q-function, $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. The value of $Q(s, a)$ represents the total collectible discounted reward starting from state s and performing action a:

$$Q(s,a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \lambda^t r_t | s_0 = s, a_0 = a\right]. \tag{2.2}$$

The values of the Q-function satisfy the Bellman equation [15]:

$$Q(s,a) = \mathbb{E}\left[r(s,a) + \lambda \max_b Q(s',b)\right], \tag{2.3}$$

where $r(s, a)$ is the earned one-step reward and s' is the next state after performing action a in state s. That is, the collectible reward from the current state equals the sum of the one-step reward and the maximum discounted collectible reward from the next state.

In Q-learning, the goal is to learn the values of the Q-function. This can be done by observing experience samples (s_t, a_t, r_t, s_{t+1}) from the environment and performing updates on the values of the Q-function in each step with a learning rate α:

$$Q'(s,a) = Q(s,a) + \mathbb{1}_{s=s_t,a=a_t}\alpha\left(r_t + \lambda \max_b Q(s_{t+1}, b) - Q(s,a)\right). \tag{2.4}$$

Once the values of the Q-function are learned for every state and action pair (s, a), the optimal policy can be derived simply by choosing the action in each step with the highest possible future reward:

$$\pi(s) = \arg\max_a Q(s,a) \tag{2.5}$$

Q-learning belongs to the family of off-policy learning algorithms. A behavior policy π_b is used to collect sample experiences during learning. It is proved that Q-learning converges to the real Q-values [16]. However, in several application scenarios, the state space or the action space can be very large (e.g. when the state is described by an image). In such cases, it is infeasible to learn all values of the Q-function. Instead, function approximators are used that try to approximate the values of the Q-function. One such algorithm is Deep Q-Networks (DQN), which uses a neural network for function approximation.

2.2.2 Deep Q-Networks

Deep Q-Networks [17] is an off-policy deep reinforcement learning algorithm. It is a model-free approach: it tries to learn an optimal policy by learning from collected experiences without building a mathematical model of the environment. DQN uses a variation of Q-learning: it tries to learn the Q-function, which is a function representing the collectible total discounted future rewards $Q(s, a)$ starting from a state s and performing action a. However, as the state space can be very high-dimensional, learning the exact values of the Q-function is impossible; instead, it tries to learn an approximation $Q^\theta(s, a)$ of the Q-function using a function approximator with parameters θ. Currently, it is done by a neural network; when the state is described in the form of an image, it can be, for example, a convolutional neural network.

The goal is to train the neural network with optimal parameters θ such that it can approximate the Q-function accurately and thus realize optimal control. To stabilize the training process, two ideas are used. First, an experience replay buffer $\mathcal{B}$ is used

to store past transitions (s, a, r, s'), and the neural network updates are performed based on minibatch samples from this buffer. This reduces the correlation among the samples used in each step of the training (which would be really high using consecutive samples) and also smoothes the changes of the data distribution over training, which is the result of the agent exploring different trajectories. Second, along with the Q-network a separate target network with parameters ϕ is also used, which is updated only periodically, thus reducing the correlation with the target.

The following loss function is used to update the parameters of the Q-network (θ):

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{B}}\left[\left(r + \gamma \max_b (Q^\phi(s', b)) - Q^\theta(s, a)\right)^2\right] \tag{2.6}$$

The target network parameters (ϕ) are updated every C_0 step with the Q-network parameters.

2.2.3 DQN extensions

To improve the stability of the training with Deep Q-Networks, two extensions are presented here.

Dueling DQN [18] proposes a new neural network architecture for training agents. This network represents two separate estimators: one for the state value function, and one for the state-action advantage function (the latter represents how advantageous it is to choose a specific action in a state compared to other actions). These estimators use the same feature extracting layers. The predicted Q values are calculated by an aggregating layer that combines the output of the two estimators.

Double DQN [19] targets the problem of overestimation. As the process of estimating action values includes a maximization step, the algorithm tends to overestimate some state action values. In this algorithm, the target

$$Y = r + \gamma Q^\phi(S_{t+1}, \arg\max_a Q^\phi(S_{t+1}, a)) \tag{2.7}$$

is replaced with

$$Y = r + \gamma Q^\phi(S_{t+1}, \arg\max_a Q^\theta(S_{t+1}, a)), \tag{2.8}$$

that is, the max operation in the target is decomposed into action selection and action evaluation with separate neural networks. That is, the greedy policy is evaluated using the online network, and the target network is used to estimate its value.

This method improves the original DQN algorithm in terms of the accuracy of the values and the quality of the resulting policy.

2.2.4 Policy gradient methods

Other than Q-learning, another family of algorithms to solve reinforcement learning problems is called policy gradient methods. In these algorithms, we use a parameterized policy π_θ. The goal is to find the optimal policy by learning the policy parameters θ directly.

For learning, a performance measure $J(\theta)$ is defined, which describes how well the policy with parameters θ performs. For example, in the case of episodic problems, this can be defined as the total collectible discounted reward under the policy π_θ from the initial state. The goal is to find a policy that maximizes this measure, hence the parameters of the policy are updated using gradient ascent in J with a learning rate α:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t). \tag{2.9}$$

Actor-critic methods extend policy gradient by using another approximator for the value function (the function $V : \mathcal{S} \to \mathbb{R}$ describing the value of each state, that is, the total collectible discounted future reward from that state) as well. The learned policy is referred to as the *actor*, and the learned value function is the *critic*.

2.2.5 Proximal Policy Optimization

Proximal Policy Optimization (PPO) [20] is a policy gradient algorithm for solving reinforcement learning problems. The main goal of the authors was to add improvements to the vanilla policy gradient method to improve the stability of training while keeping the algorithm itself simple and easily fine-tunable. PPO is an online learning algorithm, meaning that the same policy is used for generating experiences whose parameters are updated during training.

The algorithm uses an estimate of the advantage function, denoted by $\hat{A}(s, a)$, which defines how *advantageous* it is to choose action a in state s compared to other actions. The advantage can be calculated as $\hat{A}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s)$. That is, an action a being advantageous in state s means that choosing this action results in a higher future reward compared to our overall assumptions regarding the given state.

The objective function in the PPO algorithm is defined as

$$L(\theta) = \mathbb{E}\left[\min(r(\theta)\hat{A}, \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A})\right]. \tag{2.10}$$

This objective constraints the size of the policy update. The goal of this constraint is to make sure that the new policy is not too far from the old one. Here, $r(\theta)$ is defined as:

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}. \tag{2.11}$$

The ratio between the old and the new policy is limited to the interval $[1-\epsilon, 1+\epsilon]$. This clipping limits the effects of the gradient update.

The algorithm works by iterating over two phases. In the first phase, we run the policy in the environment for a fixed number of time steps to collect experiences, and compute the advantage estimates $\hat{A}_i$. In the second phase, the parameters of the policy are updated using minibatches of the collected experiences based on the defined objective function.

2.2.6 Reinforcement learning for autonomous driving

Intelligent features of self-driving vehicles have to undergo extensive safety tests before being applied in production. Applications of reinforcement learning in such scenarios are still in their early phases, as these algorithms tend to be unstable. Besides, the decisions of the agent can be difficult to explain or motivate, which would be another key aspect in a real-world application. However, there is upcoming research in applying these algorithms in this area. Some of its results from various application scenarios are presented here.

A platform for using RL algorithms for driving small-scale autonomous vehicles is presented in [21]. The DeepRacer platform makes it possible to experiment with different RL algorithms in a simulated autonomous driving environment and transfer the trained agents to the real world. They also provide a workflow for training an agent using the PPO algorithm for autonomous racing and show that this method can successfully control their racecar in their platform.

An example of autonomous road following using reinforcement learning is presented in [22]. They used the TORCS simulator [23] to train agents for lane keeping. The simulator provides the exact position and orientation of the vehicle and the surrounding objects, which is used to learn the vehicle control policy. Multiple

agents are trained and compared with discrete and continuous action spaces using different reinforcement learning algorithms.

[24] shows an example of using reinforcement learning for autonomous lane changing. In this problem, the vehicle has to accurately understand its surroundings to make a safe and efficient lane change maneuver without colliding with other vehicles.

Another scenario, namely vehicle overtaking using reinforcement learning, is presented in [25]. This is a more complex case with three phases: changing lane, passing the slower car, and returning to the original lane. They propose a multiple-goal reinforcement learning framework to cope with this scenario. Seven goals are defined to properly complete the maneuver, including lane following, collision avoidance, and keeping a steady speed.

[26] addresses the problem of navigating through unsignaled intersections. The goal, in this case, is to learn a policy that makes it possible for the cars to drive through the intersection without collision with as little latency as possible. The complexity of this case is the result of having several participants whose behavior might be unpredictable. Furthermore, occlusion can further harden the task; that is, some cars might not be visible due to being covered by other vehicles or other objects. In this paper, the DQN algorithm is used to solve the intersection problem.

2.3 Sim-to-real transfer

As reinforcement learning heavily relies on trial-and-error experiments and experience gained from previous failures and successes, a simulated environment is preferred instead of using the real-world device for training the agents. The simulator can also provide additional metrics, which can be difficult to measure precisely in the real world but can be useful for creating a meaningful reward function to train the agent. For example, in the case of autonomous lane following, it is possible to easily calculate the distance of the vehicle from the center of the lane, which is a good indicator of how accurately we are driving; but measuring this metric in the real world would be a much more difficult challenge.

However, training agents in a simulator introduces new challenges. We need to find or create a simulated environment that models the world as accurately as possible, which might require a lot of time and expertise. Also, the simulator cannot model exactly every aspect of the real world; instead, it often uses simplifications. This results in differences between the simulated and the real world. Such differences can

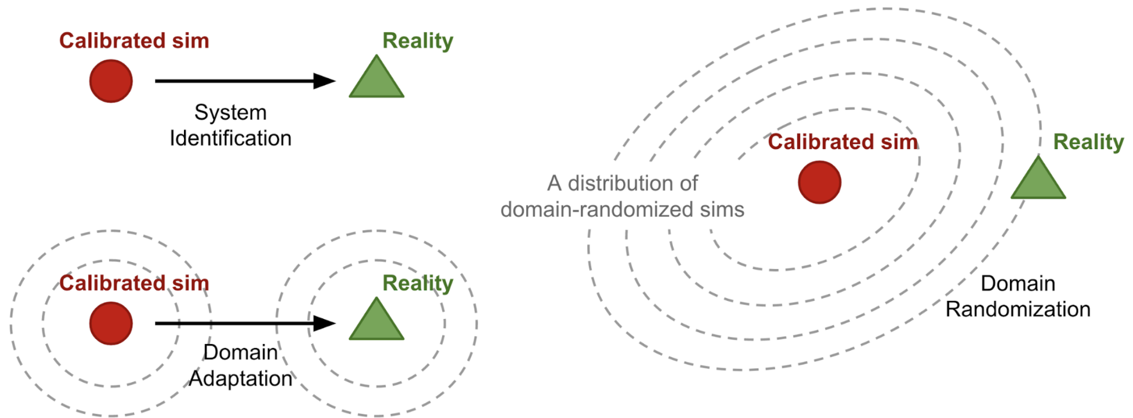


Figure 2.2: Overview of the approaches of sim-to-real transformation (source: [27]).

be, for example, the number of details, the colors of the objects, lighting conditions, dynamics of the objects, and other parameters and physical aspects.

These challenges motivate the use of sim-to-real techniques, which refer to different concepts that help to transfer the agents trained in the simulator to the real world without severe performance degradation. Sim-to-real techniques include domain adaptation, domain randomization, and system identification, which will be presented in more detail in the next sections. An overview of these methods can be seen in Figure 2.2.

2.3.1 Domain adaptation

Domain adaptation is used in a much broader spectrum of machine learning applications than just robotics and reinforcement learning. When training neural networks (or other machine learning models), we often assume that the training data has a similar distribution to the data the model will be used on. This strong assumption makes training easier; however, in many applications, it does not hold. For example, when we collect medical data of patients, we only have a small amount of training data available, which may not be an accurate reflection of the population, e.g., may contain mostly data of old people. In order to use a model trained on this data to predict the diseases of young patients, we should take into account that the model can be biased due to the differences between the training and test data. Domain adaptation techniques try to solve the challenge of training a model on one data distribution (source domain) and making it able to work on another one (target domain). For the target domain, labels are usually not or only in limited quantities

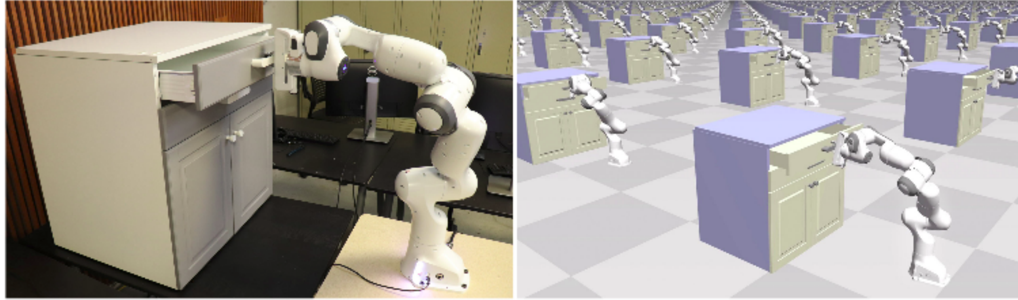
available, and thus training a model for this domain is not feasible. However, training on the source domain and transferring its knowledge to the target domain can help address this problem.

To have another example, domain adaptation can be imagined as when someone knows how to speak Italian, and would like to learn to speak Spanish. Using his/her previous Italian knowledge can significantly help to learn the new language. There are many examples in machine learning, where the source and target domain of training are difficult. For example, a face recognition system can be trained on some faces, but it has to adapt to new people as well. Similarly, a speaker-independent speech recognition system can adapt to the voices of new people. Movie recommendation systems can be trained on a set of movies, but they have to adapt to changes when new movies are released.

Formally, domains are defined as the combination of an input space $\mathcal{X}$, an output space $\mathcal{Y}$, and an associated probability distribution p. Inputs are subsets of the D-dimensional real space $\mathbb{R}^D$, and outputs can be, for example, classes: $\mathcal{Y} = \{1, \ldots, K\}$. The source domain D_S is denoted as $(\mathcal{X}, \mathcal{Y}, p_S)$ and the target domain D_T is denoted as $(\mathcal{X}, \mathcal{Y}, p_T)$. That is, input and output spaces remain unchanged during domain adaptation, only the probability distributions change.

One approach to cope with the shift between the domains is instance reweighting. Samples from the source domain that are closer to the distribution of the target domain are considered with higher weights, as they can be used more accurately for the predictions from the target domain; and other samples are considered with lower weights. [28] describes an example of this approach. They use the source domain and a small amount of data from the target domain to find the useful parts of the source domain and show that models can successfully adapt to the target domain when trained with their method. Another example of this approach can be read in [29]. They describe a complex framework, in which they combine the learning of instance weights and the usage of the Mahalanobis distance. Instance weights are learned to bridge the distribution of the source and target domains; while the Mahalanobis distance is used to maximize the distance of instances of the same class and minimize the inter-class distance for the target domain. The effectiveness of this method is proved by testing on several real-world datasets.

Another approach for domain adaptation is finding new representation for the instances of the source and target domains to reduce the distribution divergence. In other words, the samples of both domains should be transformed such that after



(a) Opening a drawer with a robot hand (source: [33]).

(b) Solving a Rubik's cube with a robot hand (source: [6]).

(c) Pushing objects to a certain location (source: [34]).

Figure 2.3: Domain randomization can be used to solve many kind of robotic tasks.

transformation, their distributions become more similar. [30] shows an example of this technique, which they apply to image classifiers trained on one dataset to recognize the instances of other datasets. [31] describes structural correspondence learning, which tries to identify correspondences among features from different domains by modeling their correlations with pivot features. This technique is applied to the field of natural language processing: they successfully transform a model trained on financial news' texts to biomedical texts. [32] uses dynamic distribution alignment for domain randomization to reach state-of-the-art results on several object recognition-based domain adaptation tasks, where the performance of classifiers is measured on a dataset other than that which was used for training.

2.3.2 Domain randomization

Another approach to cope with the problem of sim-to-real transformation is domain randomization. This relies on a simple but powerful idea: during training, we randomize some of the parameters of the simulator, thus creating different versions of the environment with different parameters. An agent trained in an ensemble of simulated environments can be expected to generalize better and work in the real

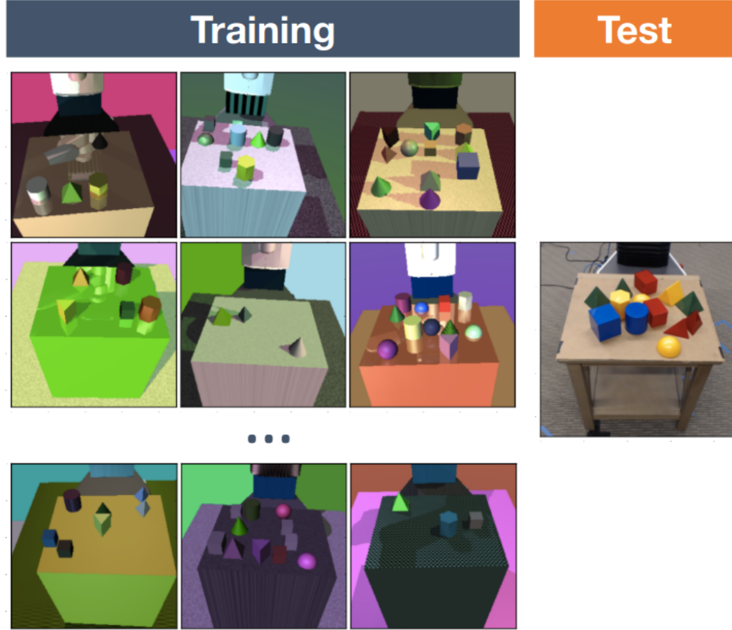


Figure 2.4: Illustration of domain randomization for robotic grasping (source: [35]).

world as well if it fits in the distribution of the training variations. The difference between the simulated and the real environment is often called *reality gap*. Some examples of using domain randomization in robotic tasks can be seen in Figure 2.3.

Training happens in an environment (e.g. simulator) over which we have full control; this is called the source domain. We would like to transfer the agent to the target domain. During training, we can control a set of M randomization parameters in the source domain e_ξ. The source domain has a configuration ξ sampled from a randomization space, $\xi \in \Xi \subset \mathbb{R}^M$. During training, episodes are collected from the source domain with randomized parameters. The policy parameter θ is trained to maximize the expected reward $\mathcal{R}(\cdot)$ average across a distribution of configurations:

$$\theta^* = \arg\max_\theta \mathbb{E}_{\xi\sim\Xi}\left[\mathbb{E}_{\pi_\theta,\tau\sim e_\xi}\left[\mathcal{R}(\tau)\right]\right] \tag{2.12}$$

where τ_ξ is a trajectory collected in source domain with configuration ξ ([27]).

[35] successfully uses domain randomization to train a neural network in a simulator that can localize objects on images, and use it to perform robotic grasping without further training on real-world images. They focus on applying domain randomization to computer vision: they train a neural network to detect the location of an object. The illustration of their method can be seen in Figure 2.4. They randomize several aspects of the simulated domain during training, for example, the number and shape

of distractor objects, colors, textures, camera position and orientation, and light sources and their characteristics. Their neural network is trained to accurately localize the target objects in each randomized image, despite the aforementioned randomizations related to the camera view. They evaluated their method by showing that by using their network to localize the target object, a robotic arm can grab it in most cases.

Domain randomization can be used not only for robotic applications but for object detection as well. [36] uses synthetic images to train a neural network for object detection. The advantage of this method over "classical" supervised learning is that it doesn't require collecting and accurately labeling real-world images, which is an expensive and time-consuming process. Instead, synthetic objects (e.g. cars) are rendered on various backgrounds along with randomized flying distractor objects. Randomized textures, lighting conditions, and viewpoints are also applied to create a synthetic dataset with a huge variety. They show that using this dataset, it is possible to reach similar accuracy on real-world images as in the case of training on those ones; and fine-tuning their model with real-world images results in a further improvement of the accuracy.

In addition to applying domain randomization to visual domains, it can also be used to randomize the dynamics of the simulator. As the physical dynamics of the real world and calibration parameters cannot be modeled completely accurately, it can help in the case of robotic applications. An example of this is described in [34]. They train a robotic arm to be able to push objects to specified locations. They show that by randomizing the simulation dynamics during training, the model can be transferred to the real world without further fine-tuning on real-world data, and it reaches similar performance in the real world as in the simulator.

It is also possible to fine-tune the simulator parameters using a few real-world roll-outs during training. An example of this is presented in [33]. The downside of randomizing simulator parameters is that we can spend a lot of time training in randomized environments which are too far from the real world, thus this part of the training will not help to transfer the policy to the real-world agent. Also, randomized parameters and randomization ranges have to be selected manually, which requires the expertise of the practitioner. The authors of this paper decided to change the simulator parameters based on real-world trials such that they are closer to the real-world domain. This helps to accelerate the learning process and resulted in successful policy transfer for two robotic tasks: opening a drawer with a robot hand

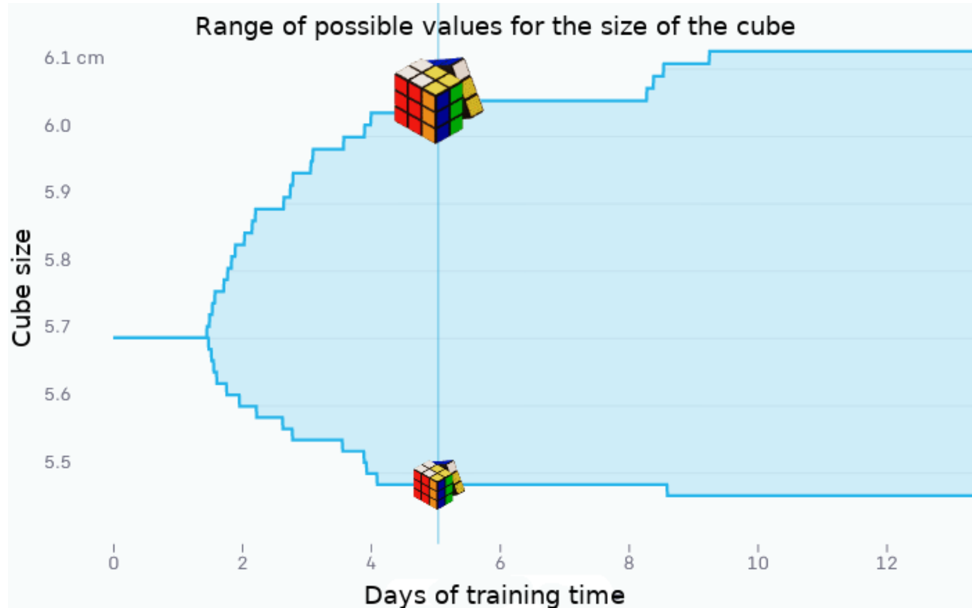


Figure 2.5: Automatic Domain Randomization automatically widens the randomization parameters during training (modified figure of [37]).

(see Figure 2.3a) and the swing-peg-in-hole task, where a peg is connected to the robot arm with a soft string, and the goal is to put it into a hole.

A more advanced technique, called Automatic Domain Randomization (ADR), is introduced in [6]. This technique is used to solve a very challenging robotic task: a human-like robotic hand is trained to solve a Rubik's cube. This task requires a policy that can control the robot very accurately. They used Kochiemba's algorithm to find an optimal solution for the given state of the cube and trained a neural network to control the robot to perform the required rotations on the cube.

The motivations for ADR are the following. Training in the environment without domain randomization results in a policy that works well in that environment, but may overfit to the specific parameters of the simulator, and thus will not work in the real world. Randomizing simulator parameters helps to solve this problem, but it hardens the training phase, and a large part of the training may be performed on randomized parameter settings that don't help to transfer to the real world. Also, selecting proper randomization settings, for example, the parameters and randomization ranges, is a difficult task. To solve these problems, ADR uses the following idea. Initially, the training is started with a single, non-randomized environment. Then, as the agent progresses, the amount of randomization is gradually increased when the agent reaches a good enough performance in the previous environments. For example, among other parameters, the size of the cube is gradually increased

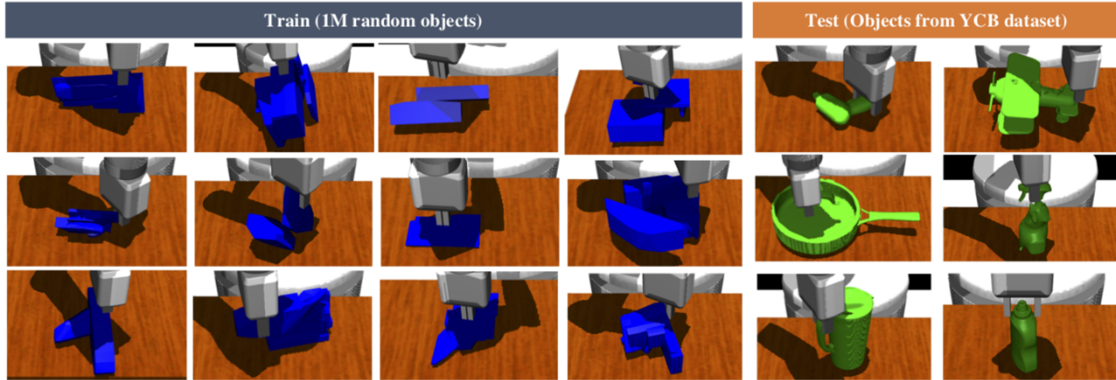


Figure 2.6: Training a model on lots of procedurally generated objects makes it able to generalize to real-world objects as well (source: [38]).

during training (see Figure 2.5). This way, the neural network has to learn to solve the task under increasingly difficult conditions. This method makes it possible to successfully solve the Rubik's cube with the real-world robotic hand with no training on real-world data and no need for having an accurate model of the real world.

The robustness of ADR is tested by applying different perturbations to the robotic hand and showing that it is still capable of solving the cube under the more difficult conditions. For example, they tied two fingers of the robot, or put a glove on it, or pushed the cube with a plush giraffe while the robot tried to solve it. These perturbations resulted in scenarios that the neural network wouldn't be able to handle without proper domain randomization techniques. Their tests confirmed that the robot was still able to solve the cube with these additional complicating circumstances. They also showed that ADR performs better than "traditional" domain randomization with fixed randomization ranges.

[38] applies the idea of domain randomization to object synthesis to perform robotic grasping. This paper focuses on the problem of generating a sufficient amount of data to train a neural network. Robotic grasping requires accurate high-quality 3D meshes of real-world objects, which are quite difficult to create. Instead, they use a simulator to create millions of unrealistic procedurally generated objects. Generating these objects is much easier than collecting accurate data of real-world objects, and with a sufficiently diverse dataset, they can train a neural network on the generated dataset that can generalize to real objects.

2.3.3 System identification

The third approach to creating successful policy transfer from the simulation to the real world is called system identification. The idea of this is to build an accurate mathematical model of the physical system [39]. This model has to be built based on observing the input and output signals of the system. The parameters of the system can be completely unknown or known up to a few parameters (physical constants).

To make the model more realistic, a very sophisticated calibration of the simulator is necessary. If we can make a precise calibration, then we can expect that the models trained in the simulator will also work in the real world. However, the disadvantage of this approach is that creating accurate calibration is expensive, and some parameters of the real-world system can change over time (due to wear-and-tear) or in different physical conditions (temperature, humidity, etc.).

[40] shows an example of creating an accurate dynamics model for a helicopter simulator. This is a challenging task, as the exact movement and dynamics of a helicopter are really complex, and are affected by a lot of factors (e.g. wind, vibration, etc.). To create a solution for this system identification challenge, they train a neural network to predict the parameters for dynamics modeling.

2.4 Preliminaries

Previous results of my research are presented in [41], where a different method was used for training the agent and transferring it to the real-world vehicle. The current thesis uses some ideas already presented in the aforementioned paper but focuses on a novel and more established method. The most important improvements are that the current method does not need any fine-tuning when using on the real-world vehicle, it performs smoother navigation and its performance is much more robust to varying environmental conditions. Furthermore, the current report also focuses on implementation and utilizing different learning algorithms for training agents.

Chapter 3

Proposed method

This section describes the details of the proposed method. Two versions of the method are presented, which differ in the learning algorithm used to train the agent. Figure 3.1 shows an overview of the method. In each iteration, the environment provides the observation S_t in the form of raw multidimensional sensory data to the agent. These observations go through essential transformation steps to form an observation sequence S'_t. N discrete actions are used for control, which are samples of the vehicle's action space $\mathbb{R}^C$. The agent receives the observation sequence, calculates a probability distribution $p_f(\cdot)$ over the possible actions, and selects A_t as the action with the highest probability. During inference, the actions are post-processed (A'_t) to stabilize the action space. The environment calculates the next state of the agent and provides a reward function R_t that quantifies the action's goodness. The agent uses this value to learn an optimal policy.

Each part of the proposed method is described in the following sections.

3.1 Environment and simulation-to-real transfer

For training the agent, an autonomous driving simulator is used. In the training phase, several parameters of the simulated environment are randomized. This technique enables the agents to learn to navigate in an ensemble of simulation environments, which results in better generalization performance.

Formally, during training, a set of M simulator parameters $\{\lambda_i | i = 1, \ldots, M\}$ are identified whose values can be changed. These parameters include both physical and observation-specific parameters.

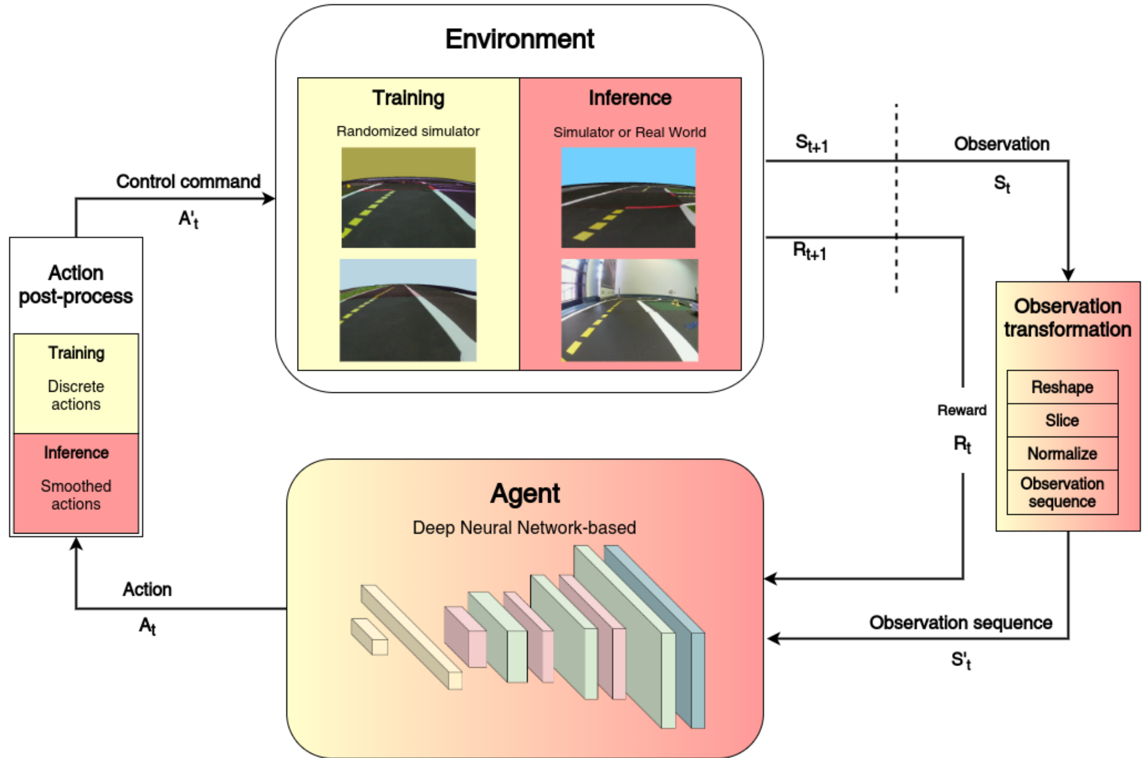


Figure 3.1: Overview of the proposed method. In each iteration, the environment provides the state to the agent in the form of raw multidimensional sensory data. During the training phase, parameters of the simulator are randomized for a better simulation-to-real transfer. The observations go through four transformation steps, and thus an observation sequence is formed, which is processed further by the agent. The agent uses a neural network to map the state to an action command. In the inference phase, the commands are post-processed for better performance.

Physical parameters include the size of the vehicle along different dimensions, the position, pitch and field of view angle of the sensor, and the speed of the vehicle. For each physical parameter λ_i, a randomization range $[P_i, Q_i]$ is determined. P_i and Q_i define the lower and upper bound of the interval of the allowed values of the parameter λ_i. The values of P_i and Q_i are specified such that their defined interval is around the approximately measured value of the parameter in the real environment. At the start of each episode, for each i the value of λ_i is sampled from a uniform distribution determined by its randomization range, i.e. $\lambda_i \sim U(P_i, Q_i)$. The values of the parameters are sampled independently from their values in the previous episodes and independently from each other.

Observation-specific parameters are specific to the type of the vehicle's sensor. For example, when the vehicles are equipped with cameras, these parameters can include the colors of the objects and global lighting conditions.

3.2 Observation transformation

The environment (which can refer both to the simulator and the sensor of the real-world vehicle) provides the state of the agent S_t in the form of a raw D-dimensional tensor $S_t \in \mathbb{R}^{d_{0,0} \times d_{0,1} \times \cdots \times d_{0,D}}$. Four essential steps are used to transform these observations. The advantage of using these steps is to make it easier for the agent to learn a good policy. The same transformation steps are used both during training and inference. Each step can be formalized as a mapping $F_i : \mathbb{R}^{d_{i,0} \times d_{i,1} \times \cdots \times d_{i,D}} \to \mathbb{R}^{d_{i+1,0} \times d_{i+1,1} \times \cdots \times d_{i+1,D}}$ that performs a transformation on the tensor. These transformation steps are:

1. **Reshape**: First, the observations are reshaped to a smaller size using interpolation. Formally, this step can be defined as a function $F_0 : \mathbb{R}^{d_{0,0} \times d_{0,1} \times \cdots \times d_{0,D}} \to \mathbb{R}^{d_{1,0} \times d_{1,1} \times \cdots \times d_{1,D}}$. The same compression rate $\delta < 1$ is used in each compression axis, i.e. $d_{1,i} = \delta d_{0,i}$ in case the observation is compressed along axis i and $d_{1,i} = d_{0,i}$ otherwise. The main advantage of this step is that the neural network can process smaller observation tensors faster, thus, both training and inference times are decreased (note that the latter is critical to realize real-time vehicle control).

2. **Slice**: Those parts of the observation that do not contain useful information for completing the given task are removed. This step can be formulated as $F_1 : \mathbb{R}^{d_{1,0} \times d_{1,1} \times \cdots \times d_{1,D}} \to \mathbb{R}^{d_{2,0} \times d_{2,1} \times \cdots \times d_{2,D}}$ where $\forall i : d_{2,i} \leq d_{1,i}$. Similarly to the previous one, this step also reduces the size of the observation, thus making training and inference faster.

3. **Normalize**: The values of the observation are rescaled from their original range to $[0, 1]$. Formally, we have $F_2 : \mathbb{R}^{d_{2,0} \times d_{2,1} \times \cdots \times d_{2,D}} \to \mathbb{R}^{d_{3,0} \times d_{3,1} \times \cdots \times d_{3,D}}$ where $\forall i : d_{2,i} = d_{3,i}$ and $F_2(x) = x / \max x$. This step makes the training more stable, as the weights and the data are in similar scale.

4. **Observation sequence**: Instead of using only the latest observation $[S_t]$ for describing the state of the agent, a sequence of k observations form the actual state S'_t. That is, we have $F_3 : \mathbb{R}^{d_{3,0} \times d_{3,1} \times \cdots \times d_{3,D}} \to \mathbb{R}^{d'_0 \times d'_1 \times \cdots \times d'_D}$, with

$d'_i = k \cdot d_{3,i}$ if the observations are stacked along axis i and $d'_i = d_{3,i}$ otherwise. This step performs a stacking of the last k observations along a specified axis. After stacking, the resulting observation sequence can describe the state of the agent more accurately than just one single observation instance.

The agent's policy network receives the resulting transformed observation sequence S'_t as input.

3.3 Training the agent

The goal of the agents is to learn a parameterized function $G_\theta : \mathbb{R}^{d'_0 \times d'_1 \times \cdots \times d'_D} \rightarrow \mathbb{R}^N$ which maps a transformed observation sequence S'_t to a probability distribution $p_f(\cdot)$ over the possible actions $a \in \mathcal{A}$. The value of $p_f(a)$ should represent how likely, or how useful, it is to choose the action a in the current state.

In this work, I used two reinforcement learning algorithms to train two agents. The first agent is trained with the Deep Q-Networks algorithm, and the second one is trained with Proximal Policy Optimization.

The agents use neural networks which are trained with the transformed observation sequences S'_t as input. For fast inference a simple neural network is utilized; it is critical to process the images in as little time as possible to successfully control the vehicle, especially in the real-world environment. The last layer of the network has N outputs, corresponding to the number of possible discrete actions.

3.3.1 Reward function

I use a continuous reward function R_t that provides valuable feedback to the agent in each time step. The purpose of such a reward function is to give continuous feedback to the agent, thus helping it to learn more effectively. The reward function considers the general properties of the agent that can be extracted from the simulation environment (e.g. position of the agent). Thus, based on the inner state of the environment, at each iteration, several metrics $\{\rho_i | i = 1, \ldots, \varrho\}$ are calculated. Such a metric can be, for example, the distance and the angle from the center of the lane or the speed of the vehicle. The reward value is computed as a function of these metrics $R_t = f_r(\rho_1, \ldots, \rho_\varrho)$.

3.4 Action space post-processing

In each time step, the agent outputs a probability distribution $p_f(a)$ over the N possible discrete actions $a \in \mathcal{A}$. Each action a is mapped to a control command $C_a \in \mathbb{R}^C$.

During training, the goal of the agent is to learn a policy to choose the optimal action sequence in an episode. At training time, at each step the agent uses the action $a^* = \arg\max_a p_f(a)$

However, during inference, the discrete actions may result in a suboptimal and fragmented action sequence. To make it more efficient, instead of selecting the action predicted with the highest probability, a combination of all actions $A'_t \in \mathbb{R}^C$ is used. To calculate this, the probability distribution predicted by the agent for the actions is used. The final action is computed as its inner product with the control commands representing the discrete actions:

$$A'_t = \sum_a p_f(a) C_a. \tag{3.1}$$

Chapter 4

Environment

In this section, I present some of the autonomous driving simulators that can be used for autonomous driving research, and introduce the details of the simulated and the real-world Duckietown environment which I used to implement the proposed method and evaluate the performance of the agents.

4.1 Autonomous driving environments

Using a self-driving simulator for training agents instead of real vehicles is a promising approach. It is much safer: for example, simulating accidents will not result in any expenses but makes the agent able to learn to avoid such situations. It is also cheaper and faster to collect data in a simulator, and the time and financial costs of labeling data can also be saved.

There are several autonomous driving environments that can be used to train models. Some of them are more complex, while some of them are simpler. Selecting a good environment that fits the needs of my current work is an important step. As the focus of this thesis is training agents with deep reinforcement learning and examining domain randomization techniques for sim-to-real transfer, I need a simulator that supports training agents with reinforcement learning (i.e. it is possible to compute a reward function that represents the precision of the vehicle's movement), and the trained models can be tested in the real world, i.e. a vehicle and an environment similar to the one in the simulator is accessible. Also, training neural networks is already a task that requires high computing power; choosing a simulator that is less complex and power-demanding makes it possible to run several experiments within a reasonable time and compare their results.

Figure 4.1: CARLA is a complex autonomous driving simulator which provides detailed, realistic environments (source: [42]).

One of the more complex autonomous driving simulators is CARLA [42]. It is an open-source simulator that provides a highly detailed and realistic environment. The simulator has a lot of features to model the real world as accurately as possible: for example, it supports day and night mode, different weather conditions (sunshine, wind, rain, snow, etc.). The environment is highly customizable: using its API, it is possible to specify custom city maps and cars, control the traffic, the behavior of the pedestrians, etc. CARLA can simulate many kinds of sensors for the cars: for example, cameras, LIDARs, etc. can be used. In addition, for the camera images, ground-truth depth estimation or semantic segmentation labels are available, which can be useful for autonomous navigation. Images from the CARLA simulator are shown in Figure 4.1.

The LGSVL Simulator [43] is another complex simulator for autonomous driving. It is also open-source and makes it possible to simulate an autonomous vehicle in a realistic 3D environment. The simulator supports multiple sensors, e.g. cameras, LiDAR, RADAR, GPS, etc. Similarly to the CARLA environment, its parameters are also customizable: for example, different weather conditions, maps, traffic and pedestrian behaviors can be configured. The simulation can be controlled through a Python API.

NVIDIA DRIVE Simulator[1] is another simulator that provides a detailed environment for real-world autonomous vehicles. It allows for creating detailed real-world

[1] `https://www.nvidia.com/en-us/self-driving-cars/drive-constellation/` (Access date: 20 May 2021)

Figure 4.2: Image of the LGSVL simulator (source: `https://github.com/lgsvl/simulator`).

maps and customizing the vehicles with different sensors. Custom traffic scenarios can be created with other vehicles, and different times (day/night) and weather conditions (sunlight, rain, fog, etc.) can be specified. The trained models can be tested in the real world using NVIDIA Constellation.

While these simulators are really great to simulate a complex urban environment, due to being very detailed, they require a lot of computing capacity. In addition, suppose a model is successfully trained in the simulator; testing it in the real world would require a car equipped with the same sensors as in the simulator. It is quite expensive, and also dangerous as an incorrect model could lead to crashes or other incidents. Thus I decided to take a look at simpler autonomous driving simulators, which can be used without a large amount of computing capacity.

In addition to highly realistic autonomous driving simulators, racing car simulators can also be considered useful for training reinforcement learning models for autonomous driving. TORCS (The Open Racing Car Simulator) [23] is an open-source racing car simulator that provides a racing environment with different kinds of cars and maps. In contrast to the previous simulators, where the goal is to drive a car precisely and according to the traffic rules on the roads, in TORCS, the goal is to drive a complete lap on the racetrack as fast as possible.

In addition to manual driving, TORCS supports controlling cars with custom algorithms. Thus, this environment was also used for autonomous racing competitions [44]. Models can be trained using sensory data (numeric metrics that describe the position of the car) or visionary input. The first method makes it easier to train

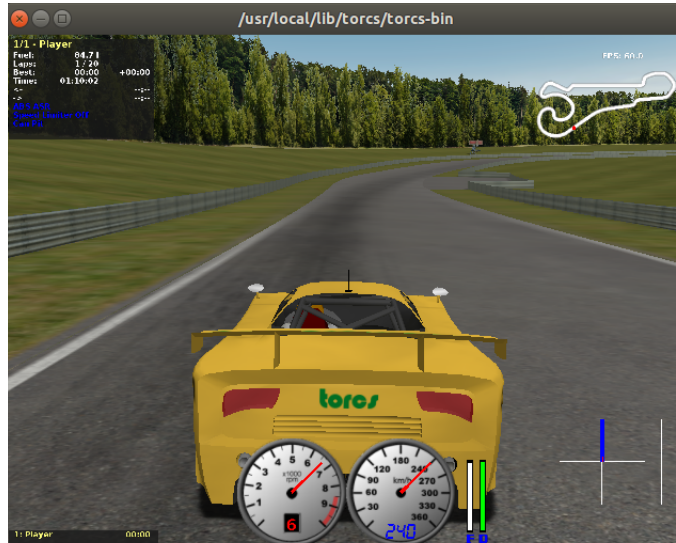


Figure 4.3: TORCS is an open-source racing car simulator that supports controlling the vehicles with custom algorithms.

reinforcement learning models, as the state has fewer dimensions, and parameters describing the state of the car in an easily utilizable way are directly available. However, the visionary input can be reproduced in the real-world environment by placing a camera on the car.

The TORCS simulator is simpler than the complex ones presented previously in this chapter, thus it can better be used to train models with reinforcement learning. However, testing its agents in the real world is still hard to realize; it could be done best with a racetrack and a racing car, which are hard to access and are very expensive.

The one environment that best suits my needs is the Duckietown environment, which is presented in more detail in the next section.

4.2 Duckietown

Duckietown[2] [9] is an educational and research platform where low cost vehicles (*Duckiebots*) travel in small cities (*Duckietowns*). The platform is made of inexpensive, off-the-shelf elements, which makes it easily accessible and ideal for education and research. The environment is highly flexible: using standardized elements, different kinds of cities can be built. The platform offers a wide range of functionalities

[2] `https://www.duckietown.org/` (Access date: 20 May 2021)

and challenges: it can be used for research related to robotics, embedded artificial intelligence, machine learning, autonomous driving, and even for cooperation between self-driving agents. All materials and codes of the platform are available open-source.

One of the biggest advantages of the Duckietown platform is that it provides an environment for self-driving vehicles that has similar scientific challenges as a more complex autonomous driving environment, but for a much lower price. This makes it widely available, and thus it is used in many well-known universities around the world (e.g. ETH Zürich, University of Montreal, etc.).

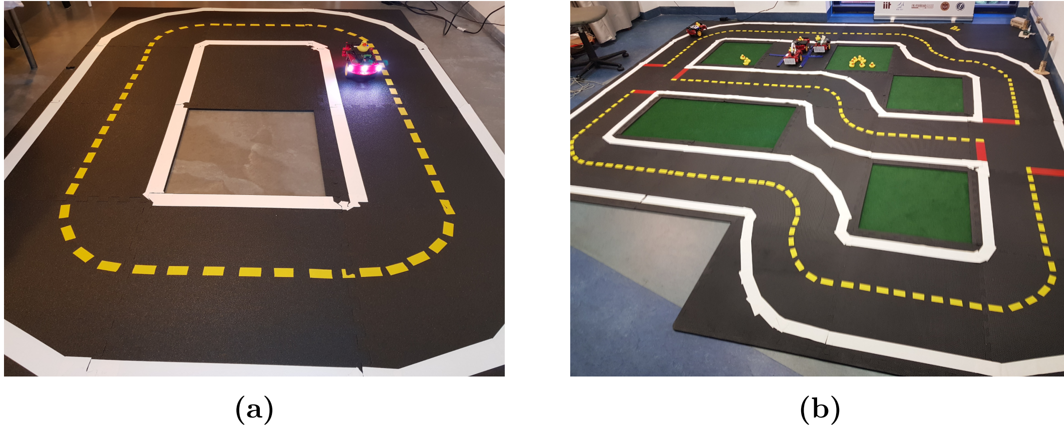
The AI Driving Olympics[3] [45] are a series of competitions on the Duckietown platform organized at the recent NeurIPS and ICRA conferences. So far, five rounds of competitions have been held; the sixth one is currently under planning. The goal of these competitions is to make it possible to compare the best-performing autonomous driving algorithms in the Duckietown environment. The competition includes many kinds of challenges in various difficulties: for example, lane following (i.e. stay in the right lane and drive as far as possible in a limited amount of time), lane following with vehicles (follow the right lane and avoid crashes with other vehicles), and lane following with vehicles and intersections (successfully drive through intersections in addition to the previous ones). In the competition, the submitted algorithms are previously tested in the simulator, and the best ones are evaluated on the real vehicles, which is the basis of the final results of the competition. As the agents are tested under well-defined metrics, the results of the competition can be used to test the effectiveness of my method and compare it to the state-of-the-art approaches on this platform.

4.2.1 Duckietowns

Duckietowns are the towns where the Duckiebots have to operate. These consist of roads, lane markings, intersections, traffic lights, signage, houses, etc. Figure 4.4 shows two example setups for the towns. The inhabitants of the town are the rubber ducks (*duckies*) that are transported by the Duckiebots; hence the naming.

Duckietowns are built of standardized road elements. These elements are straight roads, turns, 3-way, and 4-way intersections. Using these elements, different kinds

[3] https://www.duckietown.org/research/ai-driving-olympics (Access date: 20 May 2021)



(a) (b)

Figure 4.4: Example setups of the Duckietown environment: (a) shows a simpler map with a single road, while (b) shows a more complex setup with intersections in the AI Showroom of BME TMIT.

of maps can be built. Simpler maps contain only a single loop (see Figure 4.4a), while more complex setups can also contain intersections as well (see Figure 4.4b).

According to the selected setup, Duckietowns can provide different kinds of challenges for the Duckiebots. The most essential challenge is lane following on a simple map with no intersections, where the vehicle has to navigate in the right lane along the road. However, more complex challenges are also available: for example, when the map includes intersections, handling them can mean additional difficulties. Besides, when more vehicles are navigating on the map at the same time, avoiding collisions is also an important task that can be solved. More complex tasks (path planning, cooperation between vehicles) can be available as well.

4.2.2 Duckiebots

Duckiebots are vehicles that can be driven around the Duckietowns. The Duckiebot can be seen in Figure 4.5. These are small three-wheeled vehicles with a differential drive. The Duckiebots are built of inexpensive, standard parts. The "brain" of the vehicle is a Raspberry Pi 3 minicomputer, which can be used to control the vehicle. It is powered by two DC motors that drive the two wheels of the vehicle; the Duckiebot can be maneuvered by specifying wheel speed commands (two values between -1 and 1 for the two wheels). The only sensor of the vehicle is a forward-facing, wide-angle camera with a fish-eye lens, whose image has to be processed to navigate the vehicle.

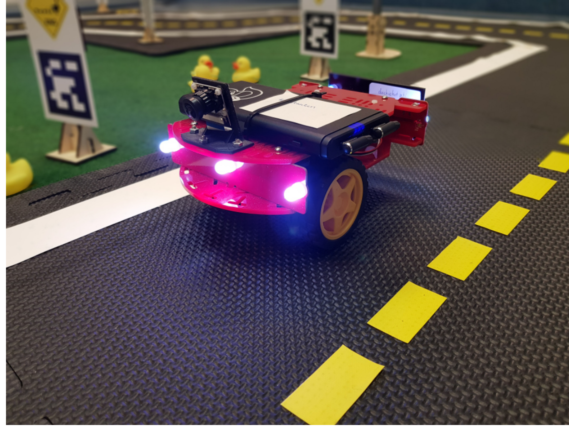


Figure 4.5: Duckiebots are small three-wheeled vehicles with a forward-facing camera. The vehicle has to be controlled by processing the camera images and giving wheel speed commands.

4.2.3 Duckietown simulator

The Duckietown platform includes a software library[4] which includes the necessary tools to interact with the infrastructure and the vehicles. This includes components for communicating with the vehicle and controlling it manually or automatically with custom algorithms; a simulated environment that can be used to train agents; baseline algorithms which usually have a really bad performance, but can be used as a good starting point to get to know the environment and try it out; and also templates for submitting models for the AI Driving Olympics.

The Duckietown simulator [46] is the part of the software library that I used for training the agents. The simulator creates a virtual view of the Duckietown environment: the virtual vehicle can be controlled with the same wheel speed commands as the real one, and the environment generates similar images to the ones that the vehicle would see in the real world. An example of an image from the simulator can be seen in Figure 4.6.

The simulator is highly customizable. For example, one can create custom maps and use them in the simulator. Also, it can be used to train models to control the vehicle more effectively than training on the real vehicle. The simulator supports creating a reward function that describes how accurately the vehicle follows the right lane.

The Duckietown environment, with all of its components, including the simulator and the easily accessible real-world parts, provides an ideal environment for me to test the proposed method and analyze the performance of the agents. A real-world

[4] `https://github.com/duckietown/` Access date: 14 April 2021

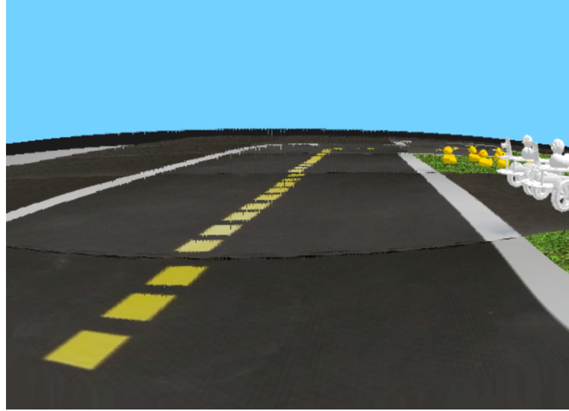


Figure 4.6: Camera image of the vehicle in the Duckietown simulator.

Duckietown environment is assembled in the BME TMIT AI Showroom, which can be used for testing agents on real vehicles. The Duckietown environment is simpler than most complex autonomous driving environments, which makes it possible to try and experiment with different methods and getting results in a reasonable time; but it still provides scientific challenges which are, in terms of difficulty, comparable to the ones of more complex environments. Thus I decided to use this environment in my work to implement the proposed method and train the agents.

Chapter 5

Implementation

In this section, I describe the details of the proposed method's actual realization. The method was used to train two agents with two deep reinforcement learning algorithms to successfully perform lane following based on visionary input in the Duckietown environment.

The camera of the vehicles provides images in a 640×480 resolution with a 30fps frame rate. The vehicle has to be controlled by specifying wheel speed commands $A_t \in [0, 1]^2$ for the two motorized wheels.

5.1 Simulation

For training, the Duckietown simulator is used. During training, several physical and observation-specific parameters of the simulator are randomized. The complete list of the randomized physical parameters and their randomization ranges are listed in A.1.

Since camera images are used as input to the vehicle, the randomized visual parameters are the parameters that change the visual appearance of the camera images. Namely, these are the color of the horizon and the position of the global lighting. The effects of randomizing the visual parameters are visualized in Figure 5.1.

The training was carried out on the track shown in Figure 5.2. This is a complex map that contains a diverse set of driving situations (e.g. long straight roads and quickly alternating turns) with the intention of preparing the agent to be able to drive on all kinds of maps during evaluation.

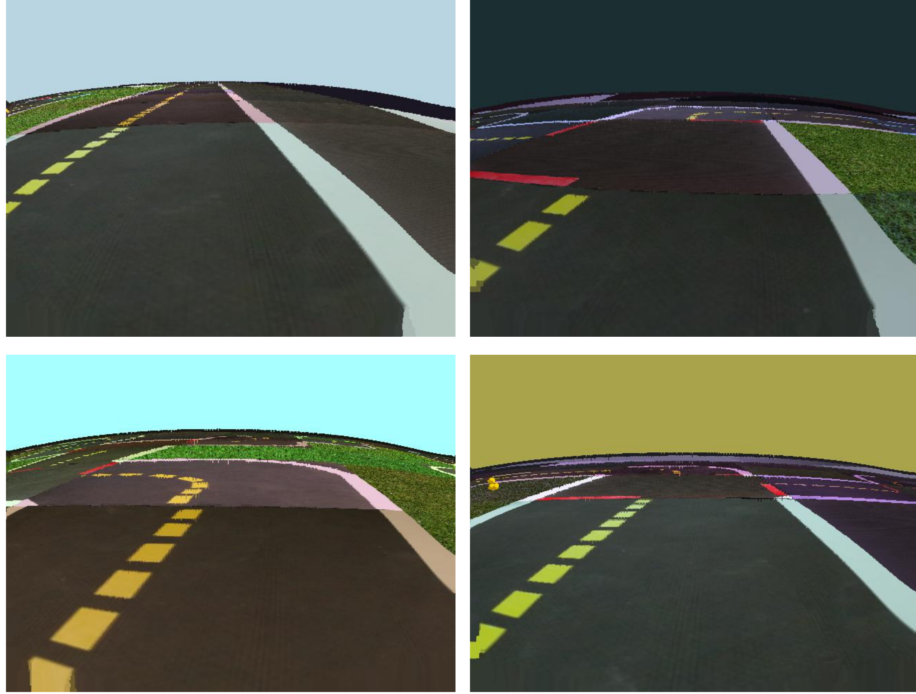


Figure 5.1: Visual parameters of the simulator, such as colors and textures, are randomized during training.

5.2 Observation transformation

The images are reshaped to a smaller resolution (80×60) and the upper one-third part of the remaining image is cropped, as this part is typically above the horizon, and thus does not provide useful information for lane following. The image's pixel's values are normalized to the $[0, 1]$ range, and the last $k = 5$ images are used to form an image sequence as an observation. This observation sequence is used as an input for the agent.

5.3 Training the agents

Two agents were trained using the proposed method with different learning algorithms to predict one of the possible actions. The action space consists of five discrete values. These actions are mapped to wheel speed commands according to Table 5.1. That is, in each time step, the agents predict a probability distribution $p_f(a)$ over the possible five actions, which describes how likely, or how useful, would it be to choose each action in the current state of the vehicle.

In both cases, the agents use a convolutional neural network with the same architecture to process the input observation sequence. It had to be designed carefully

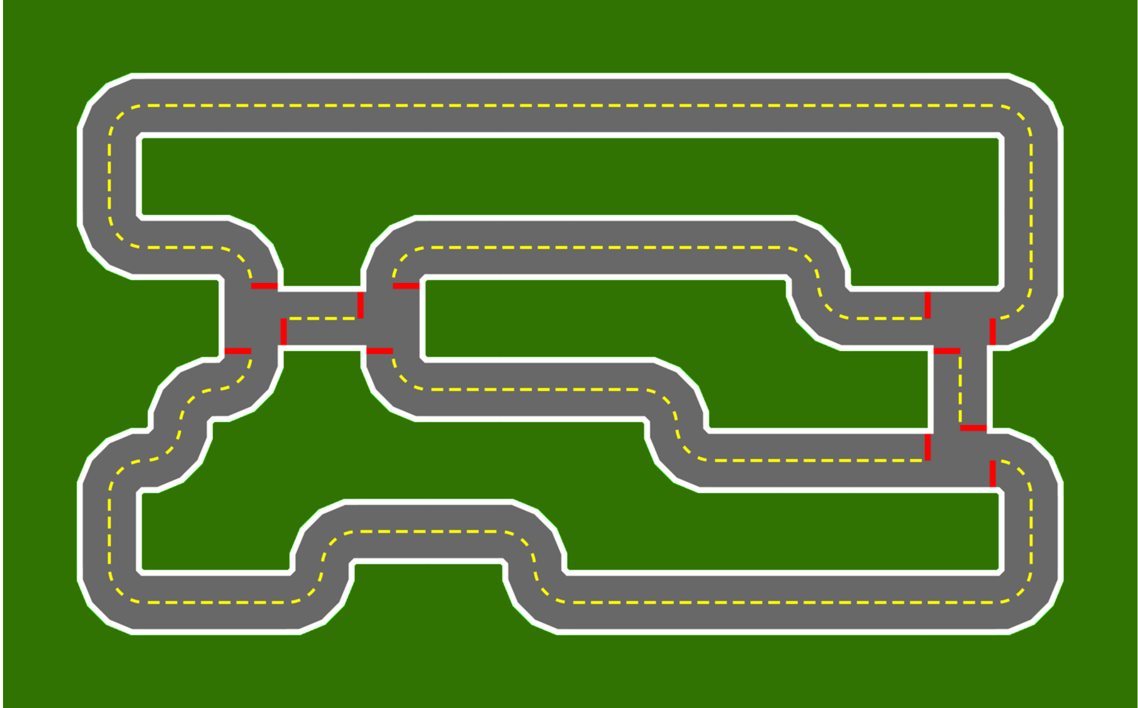


Figure 5.2: For training, a more complex map was used, with the intention of enabling the agent to learn to drive in various scenarios, e.g. long straight roads or quickly alternating turns.

such that the images can be processed in real-time in order to successfully control the vehicle. For fast inference times, a simple neural network is utilized with three convolutional layers, each followed by nonlinearity (ReLU [10]) and pooling (2×2 MaxPool) operations. The convolutional layers are followed by two fully connected layers with 128 and 5 outputs. The policy network's architecture is shown in Figure 5.3.

5.3.1 Deep Q-Networks

The first agent was trained using the Deep Q-Networks algorithm with the double Q-learning and the dueling extensions. I used the open-source implementation available in the stable baselines collection [47].

Among different hyperparameter combinations, I found these ones to be the most suitable for the training. I trained the network for 1 500 000 time steps with a batch size of 32, using a discount factor $\gamma = 0.99$, and the Adam optimizer [48] with $5 \cdot 10^{-5}$ as the learning rate. In the DQN algorithm, the size of the replay buffer was set to 150 000, and at the beginning of the training, the agent collected 10 000 time steps

Table 5.1: The discrete actions predicted by the agent are mapped to wheel speeds. (Maximum speed is 1.0.)

Action a	Left wheel speed L_a	Right wheel speed R_a
0 *Sharp left*	0.04	0.4
1 *Sharp right*	0.4	0.04
2 *Straight*	0.3	0.3
3 *Shallow left*	0.3	0.4
4 *Shallow right*	0.4	0.3

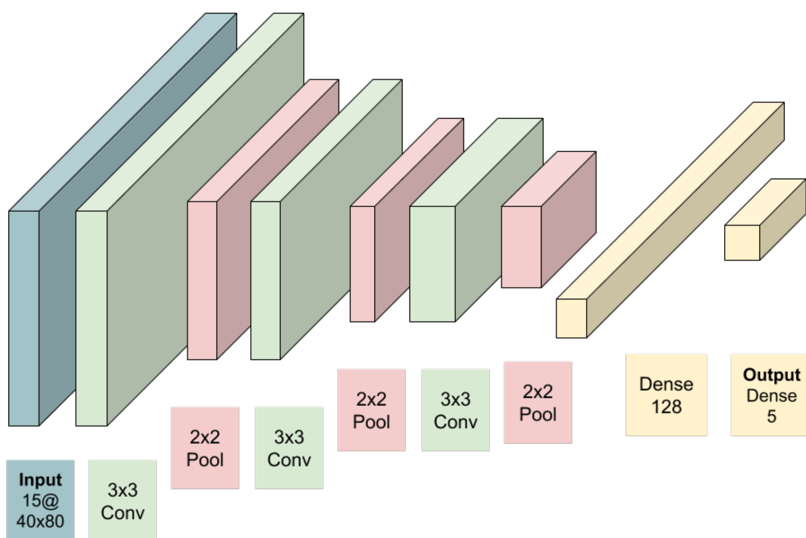


Figure 5.3: The architecture of the policy network.

experience of random actions before starting to learn. The complete training took approximately 46 hours on an NVIDIA DGX Workstation equipped with 4 V100 GPUs, an Intel Xeon E5-2698 v4 2.2 GHz (20-Core) CPU, and 256GB RAM.

5.3.2 Proximal Policy Optimization

The second agent was trained with the Proximal Policy Optimization algorithm. The network was trained for 3 000 000 time steps with a learning rate $5 \cdot 10^{-5}$. The discount factor $\lambda = 0.99$ was the same as in the case of the other agent. The training of the agent took approximately 36 hours on the same computer used for

the previous agent. Although the PPO agent was trained for more time steps than the DQN agent, these values were chosen such that the two agents are trained for a similar length of time.

5.3.3 Reward function

The purpose of the reward function is to describe how accurately the vehicle follows the lane. When it goes well in the right lane, the agent receives high rewards; when it starts to drift away from the optimal curve, it earns slightly smaller rewards; when it goes to the oncoming lane, it receives small negative rewards (penalties), and when it leaves the track, it receives high penalties (and the simulated episode also ends at this point).

The reward computed in each step uses the inner state of the environment to provide continuous feedback about the performance of the agent. The reward function is defined according to the following formula. When the vehicle is on the road, the reward is calculated as:

$$R_t = \alpha_v \cdot v \cdot \cos\beta + \alpha_d \cdot d + \alpha_p \cdot p_c, \tag{5.1}$$

where v is the speed of the vehicle in the simulator, β is the angle between the heading of the vehicle and the tangent of the optimal curve at the closest point, d is the distance from the center of the right lane, and p_c is a penalty for collisions. The coefficients α_v, α_d, and α_p are used to scale the factors of the reward. The factor $\alpha_v \cdot v \cdot \cos\beta$ encourages the agent to drive faster and follow the optimal curve. The factor $\alpha_d \cdot d$, with a negative coefficient α_d, encourages driving close to the center of the right lane. And finally, $\alpha_p \cdot p_c$ can be used to avoid collisions or driving too close to other objects. When the vehicle drives too close to an object, p_c has a negative value, otherwise, it is zero. When there are no obstacles in the track, this factor has no effect. I used the values $\alpha_v = 1$, $\alpha_d = -10$, and $\alpha_p = 40$.

When the vehicle is in an invalid position (e.g. it leaves the track), it is penalized with $-p_x = -4$, which is approximately the equal amount of penalty as the reward of driving four time steps correctly in the middle of the right lane. The simulated episode is terminated when the vehicle leaves the track.

5.4 Action post-processing

During inference, the actions are post-processed to make the movement of the vehicle smoother and less oscillating. Each discrete action a is mapped to wheel speed commands $[L_a, R_a]$ according to Table 5.1. The final speed command is calculated according to the following formula:

$$A'_t = [\sum_a p_f(a) \cdot L_a, \sum_a p_f(a) \cdot R_a] \tag{5.2}$$

.

5.5 Reference agent

To investigate the implications and show the necessity of training the agents with randomized simulator parameters, another agent was trained similarly to the DQN agent, with the only difference being that the simulator parameters were fixed to their default values during training. This agent is called the *reference agent*. I used this agent to examine the difficulties and benefits of the parameter randomization by comparing its performance to the other two agents in the simulated and the real-world environment.

Chapter 6

Evaluation and results

In this chapter, I introduce the results of training, the testing method and present the results of testing the trained agents in the simulated and the real-world environment. A video summarizing the results and showing the agents in action is available at `https://youtu.be/pkb77bc5468`.

6.1 Collected rewards

The rewards received by the agents during training per episode are shown in Figure 6.1. These values correspond to the total reward collected over each episode of training. During training, the episodes were limited to the length of 10000 time steps.

The reward curves of the DQN agent and the reference agent start with a similar steepness, but the reference agent can reach higher rewards. This is not surprising, as the latter learns only in a single environment with fixed parameters, whereas the former has to learn to drive in an ensemble of environments with randomized parameters. The PPO agent reached lower rewards than the DQN agent, despite being trained for more time steps.

6.2 Evaluation method

My goal was to create a method for reliable and stable lane following. Thus I developed the following evaluation method and used it to analyze the agents' behavior.

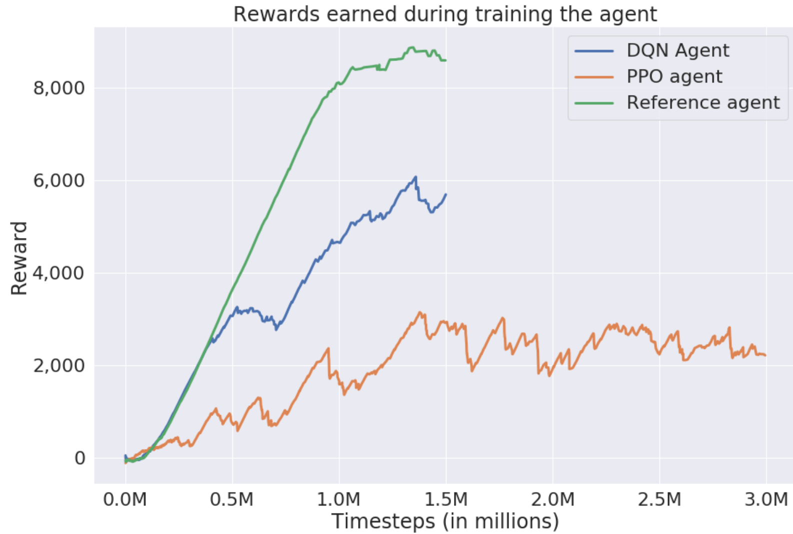


Figure 6.1: Reward earned by the agents per episodes in the training.

The performance of the trained agents was evaluated both in the simulated and the real-world environment. Several different maps were used for testing. 30 test cases were run on each map. In each test case, the vehicle was placed in a randomly selected position on the track, and the test case was considered successful if the agent was able to drive at least one complete lap in the right lane. After completing this, it has successfully driven through all parts of the track, and it can be assumed that it could do it in the following laps as well. The performance of the agent is quantified as the ratio of the successful test cases.

I used four maps for tests in the simulator. One has the same trail as the real-world map (Map #4). In each test case, the starting position and the starting angle of the vehicle were randomly selected on the drivable track. Those invalid starting positions, where the vehicle was placed such that the angle between its heading and the tangent of the optimal curve was larger than 20 degrees, and when it was placed on the side of the track facing outwards, were discarded. Each test case was run for a fixed number of time steps, which was enough to drive at least one complete lap; this limit was set to 2500, 2500, 3000, and 1200 on Map #1, Map #2, Map #3, and Map #4, respectively (30 time steps correspond to one second of driving).

The real-world tests were carried out in the setup shown in Figure 4.4a. In each test case, the vehicle was placed in a randomly selected position, heading roughly towards the correct direction (a few degrees of difference was allowed). The test cases included going around the track in both directions (i.e. clockwise and counter-

clockwise). Each test case was run for 30 seconds, which is enough to drive one complete lap in either direction.

Passing these test cases can provide a difficult challenge to the agents. To be able to do so, they have to make the correct decisions (choosing the right actions) in a long term for several consecutive time steps. While making a mistake in one of the time steps does not instantaneously result in a failure, correcting it can be difficult in the upcoming steps. In addition, making multiple wrong decisions in a short time period will lead to the failure of the test case.

To further analyze the behavior of the agent, I visualized the path of the vehicle in the simulated environment. These visualizations were created by using the precise position of the vehicle stored in the inner state of the environment.

6.3 Testing results

The results of the test cases for both the simulated and the real-world environment are summarized in Table 6.1. The table compares the results of the three agents: the DQN agent, the PPO agent, and the reference agent.

The DQN agent has very good performance in both environments, with a little lower score in the real world. However, testing in the real world poses new challenges compared to the simulated environment. It is essential to process the camera images and assign control commands with as little latency as possible, to successfully control the vehicle. In addition, as the images are processed on a computer instead of the vehicle's minicomputer, uneven network delays can also make vehicle control more difficult. I used a single computer with an Intel® Core™ i7-4500U CPU @ 1.80GHz, 12 GB of RAM, and no dedicated GPU to evaluate the performance of the agent. I measured that the method can be run with around 10 ms latency, which is sufficient for real-time vehicle control, as the camera images are arriving with a 30fps rate.

The PPO agent had slightly worse performance in both environments compared to the DQN agent. The mistakes of this agent were caused either by being started from a difficult initial location (e.g. not from the middle of the right lane, from where the agent would have to drive the vehicle back to the correct lane first), or failing at certain points of the track (e.g. making a mistake in a turn).

The performance of the reference agent in the simulated environment was very good, similarly to the DQN agent - with minor mistakes caused by failing to take one turn

in Map #1. However, this agent was unable to drive a full round on the real-world track - this is due to being trained in a single environment and possibly overfitting to certain parameters to the simulator, thus being unable to generalize to the real-world environment. This shows the importance and necessity of the parameter randomization approach during the training of the agent.

Table 6.1: Rates of successful drives on four simulated and one real-world maps.

DQN Agent			
Environment	**Test cases**		
	Total	**Successful**	**Success rate**
Simulator Map #1	30	28	93.3%
Simulator Map #2	30	29	96.7%
Simulator Map #3	30	30	100%
Simulator Map #4	30	29	96.7%
Real world	30	22	73.3%

PPO Agent			
Environment	**Test cases**		
	Total	**Successful**	**Success rate**
Simulator Map #1	30	25	83.3%
Simulator Map #2	30	18	60%
Simulator Map #3	30	20	66.7%
Simulator Map #4	30	24	80%
Real world	30	16	53.3%

Reference Agent			
Environment	**Test cases**		
	Total	**Successful**	**Success rate**
Simulator Map #1	30	24	80%
Simulator Map #2	30	29	96.7%
Simulator Map #3	30	27	90%
Simulator Map #4	30	29	96.7%
Real world	30	0	0%

The visualizations of the paths driven by the DQN agent for the four maps used in the simulator are shown in Figure 6.2. The figures show several successful and one unsuccessful test case. The initial position is marked with a red circle. In some cases (e.g. (e), (i)) the vehicle was started from the oncoming lane, but it was still able to navigate back to the right lane and continue driving there. In the unsuccessful test

case (c), the vehicle was started from a position from where it is difficult to navigate back to the right lane.

The paths of the PPO agent in the tests are visualized in Figure 6.3. The figures show several successful and two unsuccessful attempts. In most cases, the agent was able to successfully follow the right lane. In the unsuccessful test scenarios, the vehicle was started from a difficult position or from the oncoming lane, from where the agent was not able to recover and navigate back to the right lane.

6.3.1 Effects of action post-processing

I examined the effects of the proposed action post-processing method in the case of the DQN agent to show that it results in smoother vehicle control. The following experiment was carried out in the simulator. I used a map with only a long straight road and placed the vehicle at the beginning of the road. Then, the agent was instructed to perform lane following for 2000 time steps with and without post-processing the actions. I measured the mean absolute difference between the speeds of the wheels during the runs. The average of these metrics over 15 experiments were calculated. This metric measures how oscillating the vehicle's movement is. If the agent has to make more corrections (small turns) to be close to the center of the lane, it results in a higher average difference between the speeds of the wheels.

Without action post-processing, the mean absolute difference between the wheels' speeds was 0.169; with action post-processing, it was reduced to 0.026. This experiment confirms that the vehicle's movement became much smoother by post-processing the actions.

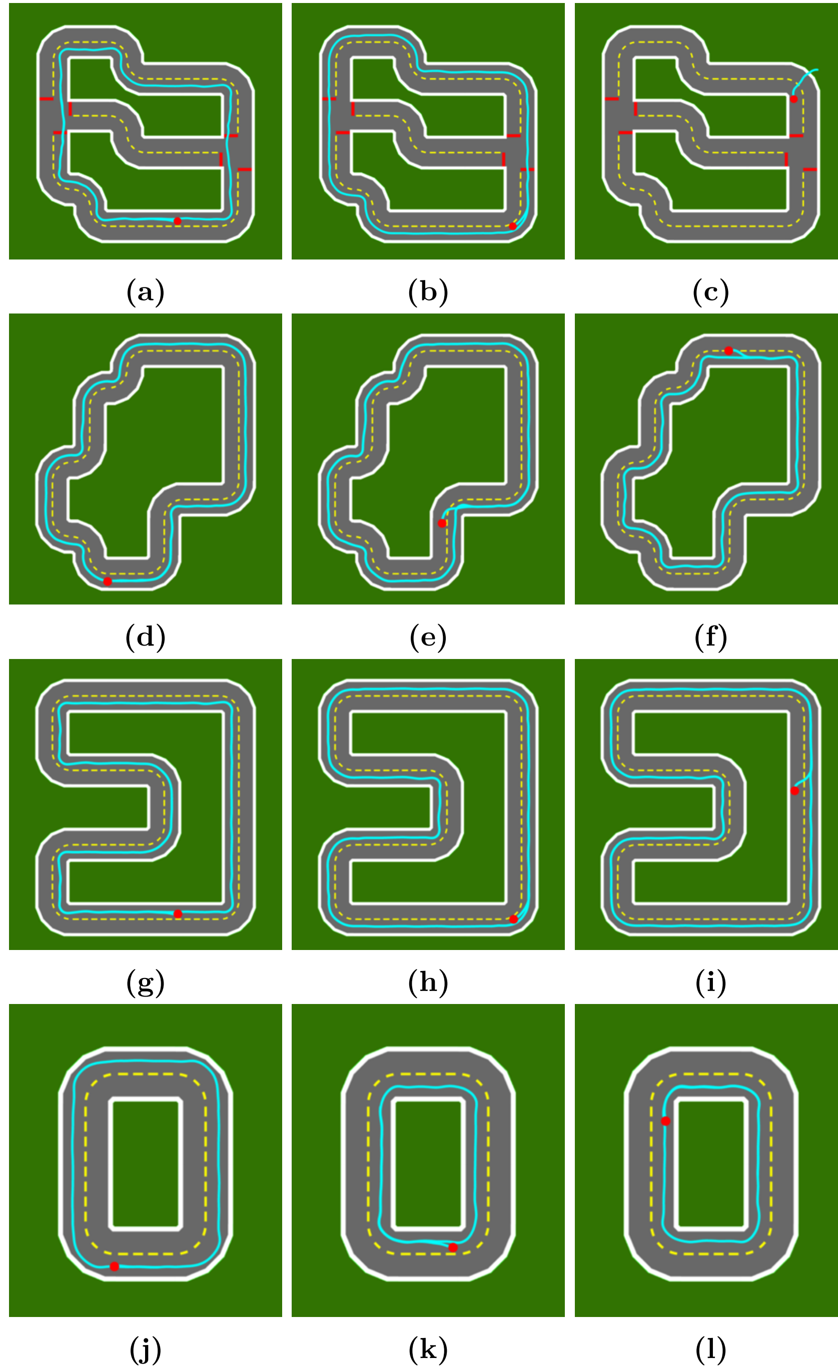


Figure 6.2: DQN Agent navigation patterns in the simulator. Figures (a)-(c), (d)-(f), (g)-(i) and (j)-(l) show the paths of the vehicle in Simulator Map #1, Map #2, Map #3, and Map #4, respectively. The starting locations of the vehicle are marked with a red circle, and its paths are drawn with a blue line. While most pictures show examples of successful test cases, figure (c) shows an example of a failed test case.

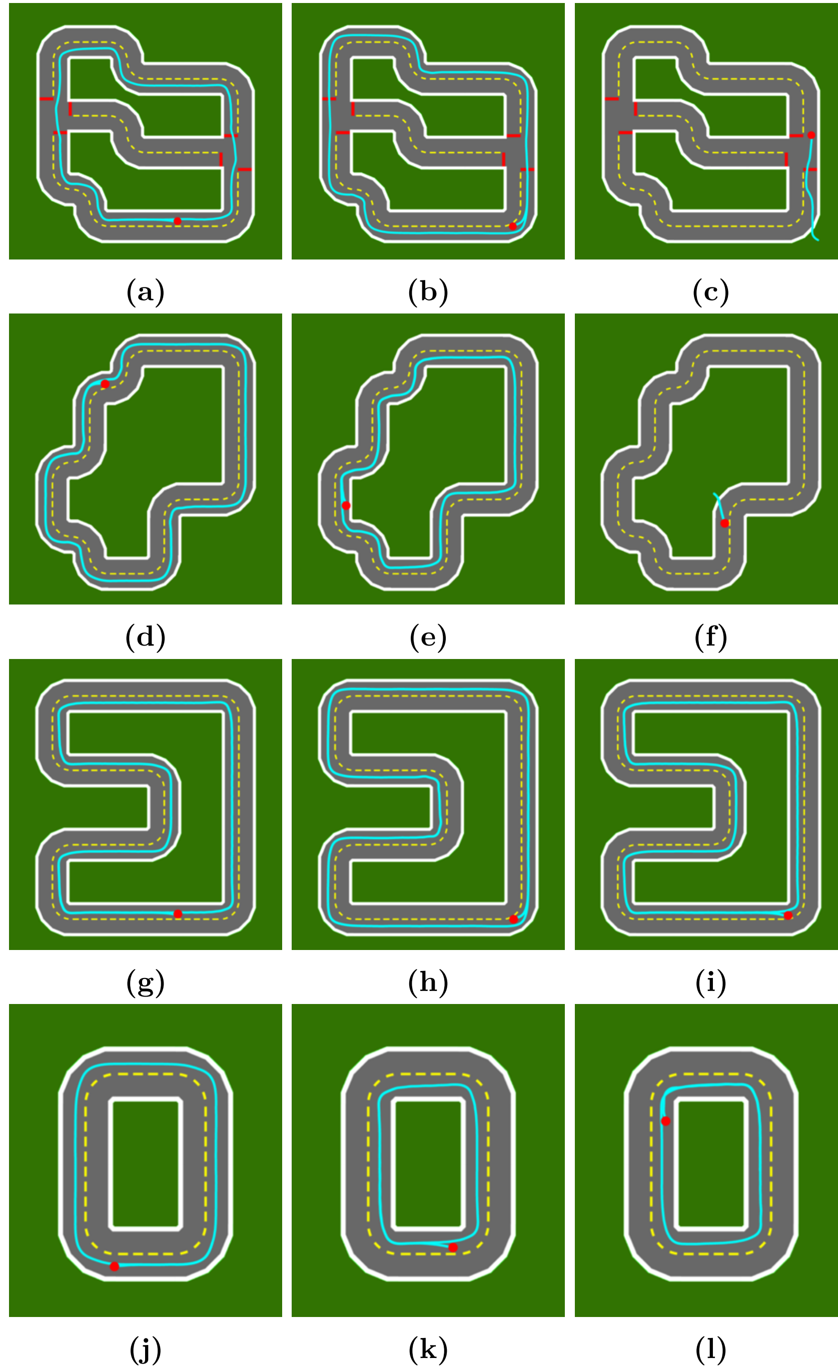


Figure 6.3: PPO Agent navigation patterns in the simulator. Figures (a)-(c), (d)-(f), (g)-(i) and (j)-(l) show the paths of the vehicle in Simulator Map #1, Map #2, Map #3, and Map #4, respectively. The starting locations of the vehicle are marked with a red circle, and its paths are drawn with a blue line. While most pictures show examples of successful test cases, figures (c) and (f) show examples of failed test cases.

Chapter 7

Summary

In this thesis, I addressed the challenge of autonomous lane following using deep reinforcement learning. I developed a complete pipeline for training agents in a simulated autonomous driving environment. I proposed a method for successfully transferring the agents to the real-world environment without fine-tuning on real-world data. I implemented the presented method in the Duckietown environment and trained two agents using different learning algorithms: Deep Q-Networks and Proximal Policy Optimization. I introduced an evaluation method and used it to test and compare the performance of the agents. The trained agents were able to successfully perform the lane following task both in the simulated and the real-world environment in most cases.

Seeing the successful results of this method in the Duckietown environment arises the question of whether it could be used in other autonomous driving environments as well.

The method was designed to be generally applicable to similar environments. It does not contain any parts that are specific to the Duckietown environment. That is, it does not involve, for example, hand-engineering features specific to one environment (e.g. performing lane segmentation based on the appearance of the Duckietown environment and training a model on the segmented images).

Although the method was only tested in the Duckietown environment, I believe that it can be used to perform lane following in similar autonomous driving environments as well. In the case of environments being more similar to the Duckietown environment (e.g. the DeepRacer platform), the method should be applicable with only minor modifications. Such modifications can include the physical properties of the environment (e.g. the setup and the properties, including the physical parame-

ters and the sensor type, of the controlled vehicle, and the hyperparameters of the training algorithms).

In the case of more complex environments (e.g. the CARLA autonomous driving simulator), the core of the method should still be applicable, but broader modifications might be necessary. For example, in more complex environments, the surroundings of the vehicle might be more compound, meaning that a neural network with a simple architecture proposed in this thesis might not be able to accurately estimate the location of the vehicle. Thus using a more complex network might be favored, which also requires more computing resources both for training and inference. Driving vehicles with a steering wheel instead of a differential drive also requires changing the agent's action space. That is, modifying the action space that fits such vehicles is required.

Regarding future work, this paper already presents a complete pipeline for training agents for autonomous lane following. A journal paper describing the method, supplementing it with further analysis, and proving its robustness with extreme test scenarios, is currently under preparation. The source code for training the agents will be made open-source along with the publication of the paper.

In the following, I present some ideas for extending the method for different use cases and more complex application scenarios.

The current method focuses on realizing a method for stable and reliable lane following. This can be extended with a focus on driving at higher speeds. Such a method could be successfully used in the AI Driving Olympics, where the qualification criterion is defined as fast and accurate lane following in the simulator, and the final rankings are determined based on how precisely and fast the agents can drive the real-world vehicle.

The current method, which successfully realizes lane following, can be the basis of more complex applications in autonomous systems, such as collision avoidance and path planning. In the former case, obstacle detection methods can be used to monitor the surroundings of the vehicle and detect possible collisions, e.g. with other vehicles, pedestrians, or other objects. In the latter, the agent can be expected to drive to a specific location in the shortest possible time on a more complex setup (i.e. using a track with several intersections and possibly also with other vehicles).

Future work can also involve investigating the possibilities of training the agent to be able to recover from failures (e.g. leaving the road or colliding with other objects). One possible solution for this is making it able to drive backward and find its way

back to the track. This way, the agent could detect when it gets into an invalid location (from where normally it is impossible to continue driving on the road) and correct its mistake.

Acknowledgements

The author is pleased to thank his supervisors, Bálint Gyires-Tóth, and Róbert Moni, for their continuous support and advice, which greatly contributed to the successes of this work.

This work was supported by the ÚNKP-20-2 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

The author would like to express his gratitude for Nokia Bell Labs Hungary, for the continuous support of his studies in the IMSc program with the Nokia Bell Labs IMSc Scholarship.

The author is also thankful for Continental AG for supporting the research with scholarship and providing the research environment through the Professional Intelligence for Automotive project.

Bibliography

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. DOI: 10.1109/CVPR.2016.90.

[2] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017. DOI: 10.1109/CVPR.2017.243.

[3] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[4] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[5] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

[6] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

[7] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.

[8] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojtek Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.

[9] Liam Paull, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Yajun Fang, et al. Duckietown: an open, inexpensive and flexible platform for autonomy education and research. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1497–1504. IEEE, 2017.

[10] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, 2010. ISBN 9781605589077.

[11] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28, pages 91–99. Curran Associates, Inc., 2015.

[12] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. DOI: `10.1109/5.726791`.

[13] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition

Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. DOI: 10.1007/s11263-015-0816-y.

[14] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[15] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[16] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8 (3-4):279–292, 1992.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[18] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003. PMLR, 2016.

[19] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, pages 2094–2100, 2016.

[20] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[21] Bharathan Balaji, Sunil Mallya, Sahika Genc, Saurabh Gupta, Leo Dirac, Vineet Khare, Gourav Roy, Tao Sun, Yunzhe Tao, Brian Townsend, et al. Deepracer: Autonomous racing platform for experimentation with sim2real reinforcement learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2746–2754. IEEE, 2020.

[22] Ahmad El Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. End-to-end deep reinforcement learning for lane keeping assist. *arXiv preprint arXiv:1612.04340*, 2016.

[23] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net*, 4(6):2, 2000.

[24] Pin Wang, Ching-Yao Chan, and Arnaud de La Fortelle. A reinforcement learning based approach for automated lane change maneuvers. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1379–1384. IEEE, 2018.

[25] D. C. K. Ngai and N. H. C. Yung. A multiple-goal reinforcement learning method for complex vehicle overtaking maneuvers. *IEEE Transactions on Intelligent Transportation Systems*, 12(2):509–522, 2011. DOI: 10.1109/TITS.2011.2106158.

[26] D. Isele, R. Rahimi, A. Cosgun, K. Subramanian, and K. Fujimura. Navigating occluded intersections with autonomous vehicles using deep reinforcement learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2034–2039, 2018. DOI: 10.1109/ICRA.2018.8461233.

[27] Lilian Weng. Domain randomization for sim2real transfer. *lilianweng.github.io/lil-log*, 2019. URL http://lilianweng.github.io/lil-log/2019/05/04/domain-randomization.html.

[28] Wenyuan Dai, Qiang Yang, Gui-Rong Xue, and Yong Yu. Boosting for transfer learning. In *Proceedings of the 24th International Conference on Machine Learning*, pages 193–200, 2007.

[29] Yonghui Xu, Sinno Jialin Pan, Hui Xiong, Qingyao Wu, Ronghua Luo, Huaqing Min, and Hengjie Song. A unified framework for metric transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 29(6):1158–1171, 2017.

[30] Basura Fernando, Amaury Habrard, Marc Sebban, and Tinne Tuytelaars. Unsupervised visual domain adaptation using subspace alignment. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2960–2967, 2013.

[31] John Blitzer, Ryan McDonald, and Fernando Pereira. Domain adaptation with structural correspondence learning. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 120–128, Sydney, Australia, 2006. Association for Computational Linguistics.

[32] Jindong Wang, Wenjie Feng, Yiqiang Chen, Han Yu, Meiyu Huang, and Philip S Yu. Visual domain adaptation with manifold embedded distribution alignment. In *Proceedings of the 26th ACM International Conference on Multimedia*, pages 402–410, 2018.

[33] Yevgen Chebotar, Ankur Handa, Viktor Makoviychuk, Miles Macklin, Jan Issac, Nathan Ratliff, and Dieter Fox. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8973–8979. IEEE, 2019.

[34] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3803–3810, 2018. DOI: 10.1109/ICRA.2018.8460528.

[35] 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. DOI: 10.1109/IROS.2017.8202133.

[36] Jonathan Tremblay, Aayush Prakash, David Acuna, Mark Brophy, Varun Jampani, Cem Anil, Thang To, Eric Cameracci, Shaad Boochoon, and Stan Birchfield. Training deep networks with synthetic data: Bridging the reality gap by domain randomization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 969–977, 2018.

[37] Solving rubik's cube with a robot hand, Oct 2019. URL https://openai.com/blog/solving-rubiks-cube/.

[38] Josh Tobin, Lukas Biewald, Rocky Duan, Marcin Andrychowicz, Ankur Handa, Vikash Kumar, Bob McGrew, Alex Ray, Jonas Schneider, Peter Welinder, et al. Domain randomization and generative models for robotic grasping. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3482–3489. IEEE, 2018.

[39] Lennart Ljung. *System Identification*, pages 163–173. Birkhäuser Boston, Boston, MA, 1998. ISBN 978-1-4612-1768-8. DOI: 10.1007/978-1-4612-1768-8_11. URL https://doi.org/10.1007/978-1-4612-1768-8_11.

[40] Ali Punjani and Pieter Abbeel. Deep learning helicopter dynamics models. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3223–3230. IEEE, 2015.

[41] Péter Almási, Róbert Moni, and Bálint Gyires-Tóth. Robust reinforcement learning-based autonomous driving agent for simulation and real world. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

[42] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[43] lgsvlsim. Lgsvl simulator, Jan 2020. URL `https://www.lgsvlsimulator.com/`.

[44] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. Simulated car racing championship: Competition software manual. *arXiv preprint arXiv:1304.1672*, 2013.

[45] Julian Zilly, Jacopo Tani, Breandan Considine, Bhairav Mehta, Andrea F Daniele, Manfred Diaz, Gianmarco Bernasconi, Claudio Ruch, Jan Hakenberg, Florian Golemo, et al. The ai driving olympics at neurips 2018. In *The NeurIPS'18 Competition*, pages 37–68. Springer, 2020.

[46] Maxime Chevalier-Boisvert, Florian Golemo, Yanjun Cao, Bhairav Mehta, and Liam Paull. Duckietown environments for openai gym. `https://github.com/duckietown/gym-duckietown`, 2018.

[47] 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.

[48] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

Appendix

A.1 Randomization values

DR values and their randomization ranges are described in Table A.1.

Table A.1: Randomized simulator parameters and randomization ranges.

Parameter (λ_i)	Lowest (P_i)	Highest (Q_i)
Speed multiplier	0.5	2.0
Camera pitch angle	15,96°	22.98°
Camera FOV angle	62.5°	90°
Camera distance from floor	0.090 m	0.130 m
Camera distance from center of rotation	0.055 m	0.079 m
Distance of wheels	0.093 m	0.102 m
Robot width	0.136 m	0.150 m
Robot length	0.164 m	0.180 m
Robot height	0.109 m	0.120 m