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

Real-world autonomous driving using deep reinforcement learning and domain randomization

Scientific Students' Association Report

Author:

Péter Béla Almási

Advisor:

Dr. Bálint Gyires-Tóth

2020

Contents

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, továbbá idősorelemzési feladatokban is kiemelkedőnek bizonyultak. A mélytanuló modellek számos 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 intelligens 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.

A mély megerősítéses tanulás használata sok esetben még nagyobb kihívást jelent olyan feladatok esetén, amelyek részben vagy egészben valós környezetben működnek - például robotokat vagy járműveket szeretnénk vezérelni. Ilyen feladatoknál jellemzően az intelligens ágenseket egy szimulátorban tanítják, majd a kész modellt "átültetik" a valós robotra. A megerősítéses tanulás alkalmazása önvezető járművek esetében már szimulátorban is nehéz kihívás, hiszen ezek az algoritmusok sok esetben instabilak, és nem rendelkeznek minden esetben 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. Megoldásom alapján mély megerősítéses tanítás segítségével lehetséges olyan intelligens ágenseket kialakítani, melyeket szimulációs környezetben tanítok be, és a valós környezetben, valódi robotokon futtatva is hasonló pontosságot nyújtanak - a valós környezetből vett tanító minták nélkül.

A kidolgozott módszer robusztusságát olyan extrém körülmények között végzett tesztekkel igazolom, amelyekkel tanítás során az ágens nem találkozott explicit módon. A szimulációs környezetben, nappali körülmények között tanított ágens a valós környezetben akár éjszaka is képes közlekedni, illetve szabálytalan pozícióból indítva is vissza tud térni a megfelelő útsávba. Az eredményeket egy demonstrációs videóban is bemutatom, amelyre a hivatkozás megtalálható a dolgozatban.

Abstract

Deep Neural Networks have received great attention recently. These models have been successfully applied to reach *state-of-the-art* results in various application scenarios: for example, image recognition, object localization, speech recognition and synthesis, natural language processing, time series analysis, etc. These models can even solve specific tasks on a superhuman level.

Deep Reinforcement Learning (DRL) is a field of machine learning which enables intelligent software agents in an environment to attain their goal. They utilize deep neural networks to learn the best possible actions in each state. This technique has been successfully applied to beat world champions in different board and computer games, for example, Go or StartCraft II.

However, solving tasks involving real-world devices, i.e. 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. I describe the details of training self-driving robots in the Duckietown simulation environment with deep reinforcement learning. I develop a method to effectively transfer agents from the simulator to the real robots. As a result, the agents are able to drive autonomously in the real-world environment without further training on real-world data. I evaluate the robustness of the proposed method with extreme test cases that were not involved in the training process.

The results show that the agent taught in the simulator in daytime conditions can travel in the real environment even at night or can return to the appropriate lane when started from an irregular position. Besides numerical results, a demonstration video also presents the solution, which is referred to in this document.

Chapter 1

Introduction

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

Among its subfields, deep learning has become a very actively researched area recently. Thanks to the recent advances in the development of GPUs (Graphics Processing Units), the computing power available via cloud services, algorithmic developments, and the availability of big databases, training of deep neural networks have 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 abilities.

Deep learning shows 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 the development of self-driving cars. However, the problem of autonomous driving is far from being completely solved, thus, it is an active research area 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 an environment and tries to reach a specified goal by taking 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 [8].

Nonetheless, using DRL algorithms to solve real-world robotics or autonomous driving tasks seems to be a more difficult challenge. The desired approach would be to train an agent in a simulator and then use it in the real world. The simulator has lots of advantages: it is much safer, it makes collecting an unlimited amount of labeled data possible, and is a lot cheaper than using real robots. 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 carefully designed, to be able to run with very little latency, for instance. Sim2Real refers to the concept of transferring robotics skills learned in a simulator to real robots. 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.

The Duckietown environment [9] is an educational and research platform where small three-wheeled robots can be used to perform autonomous driving tasks, for example, lane following, or collision avoidance. The environment is open, inexpensive, and highly flexible. The robots' only sensor is a wide-angle monocular camera. The most essential task is to process the image stream of the camera and perform autonomous lane following based on this information.

In this work, I propose a novel method for training autonomous driving agents in an autonomous driving simulator and transferring these agents to real-world vehicles. In the proposed method, a deep reinforcement learning-based algorithm is used to train a robot to perform lane following based on high-dimensional camera input. I analyze and develop Sim2Real methods to make it possible to transfer the model trained in the simulator to the real robot. For the evaluation of the proposed method, the Duckietown environment is used. The trained agent is capable of performing lane following both in the simulator and the real-world Duckietown environment successfully. My method is designed such that it can be run in real-time on an average computer with limited hardware resources[1].

[1] Intel®Core™i7-4500U CPU @ 1.80GHz without dedicated graphics accelerator

I justify the robustness of the method with test cases carried out in extreme scenarios that the agent was not taught explicitly. It is able to drive in real-world environments at night, despite being taught only in daytime conditions in the simulator. These conditions provide significantly different visual inputs for the agent. In addition, the agent can recover from invalid starting locations, such as the oncoming lane, and navigate back to the right lane. These extreme test cases are also presented in a demonstration video, which is referred to in this document.

The rest of this document is organized as follows. Chapter 2 introduces the theoretical background and the relevant methods of deep learning, reinforcement learning, and sim-to-real transformation. The proposed method is described in Chapter 3. The Duckietown environment, which I used to test my method, is introduced in Chapter 4 along with the details of the implementation. I evaluate my method and present the results in Chapter 5, and give a summary in Chapter 6.

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, focusing on its applications in 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 by experience. 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 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.

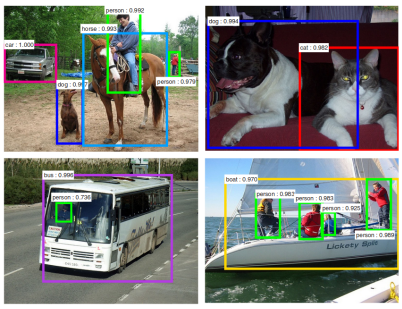
Machine learning techniques can be categorized into three different approaches. In the case of *supervised learning*, the algorithm is presented with both training data

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 input and, for example, detect anomalies or cluster the data based on similarities. In the case of *reinforcement learning*, an agent interacts with a dynamic environment 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 to take optimal actions to reach its goal. Reinforcement learning is used, for example, to train models to learn 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 matrices or multidimensional tensors. These 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; 3-D convolutions can be used for video processing; 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 we increase the amount of training data. 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 improvements in 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

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. It 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 greatest image recognition algorithms. The fact that this competition is being won by convolutional neural networks since 2012 (e.g. [1]) strengthens that CNNs are able to produce *state-of-the-art* results for image recognition.

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 location needs 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

Reinforcement learning is an area of machine learning where an agent interacts with a dynamic environment to reach a specified goal by learning from the feedback of the environment. This is typically used when we would like to train an agent to learn to play board games or computer games. An overview of the interaction between the agent and the environment can be seen in Figure 2.2.

More formally, in the current setting, I consider an agent that interacts with an environment $\mathcal{E}$ in discrete timesteps. As it can be seen in Figure 2.2, in each timestep t, the environment provides the agent its state S_t (in my work, it is in the form of a high-dimensional RGB image). The agent chooses its action A_t from a set of possible actions $\mathcal{A} = \{A_1, ..., A_k\}$ according to a policy $\pi(A|S)$. The environment computes a reward R_t based on the selected action in the given state, which describes how *good* the chosen action was. The environment also provides this reward value to the agent in the next step. The goal of the agent is to maximize the total reward

$$R_t = \sum_{u=t}^{T} R_u \gamma^{u-t}, \tag{2.1}$$

where γ is the discount factor ($\gamma \in [0, 1]$), and T is the timestep at which the episode terminates. The goal of the agent is to find a policy π^* which maximizes the expected reward when it is used to determine the next actions:

$$\pi^* = \arg\max_{\pi} \mathbb{E}[R_t^{\pi} | S_t = S, A_t = A]. \tag{2.2}$$

Reinforcement learning algorithms have been successfully used in recent years to surpass human players in complex board games and computer games. One of the most well-known recent successes of RL was making an agent that was able to beat the world champion in the game Go [14]. This agent, in contrast to most algorithms playing board games, uses no expert knowledge learned from skilled players. It learns to find winning strategies by playing against itself millions of times and uses the results of these games to find the best steps in each situation. With this technique,

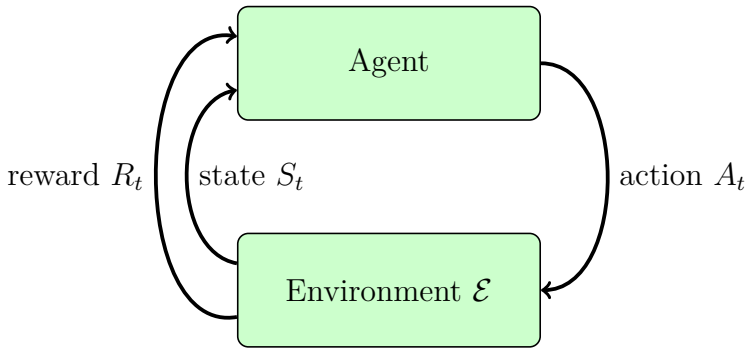


Figure 2.2: Interaction between the agent and the environment in reinforcement learning.

the agent can learn winning strategies that were previously undiscovered by human players.

In addition to board games, these algorithms can also be taught to play complex computer games. AlphaStar [15] was trained to play the StarCraft II game, and it successfully overpowered human players. The algorithm learned to play among similar constraints as human players do (e.g. viewing the game world through a camera, and limiting the frequency of actions). The authors used several machine learning techniques (e.g. self-play via reinforcement learning, neural networks, imitation learning, and multi-agent learning) to learn the best policies. OpenAI Five [16] reached expert human-level performance in the Dota 2 game. This also demonstrates that self-play reinforcement learning can be used to reach superhuman performance on complex tasks.

Since the observations in my work are represented in the form of high-dimensional RGB images, I decided to use the Deep Q-Networks algorithm to find the optimal policy that can control the robot. This algorithm was designed to process image inputs and take actions to learn to play computer games based on this information, which, on a high-level view, is similar to my current task.

2.2.1 Deep Q-Networks

Deep Q-Networks (DQN) [17] [18] is one of the most well-known reinforcement learning algorithms. This algorithm was originally developed to solve many different versions of the classic Atari 2600 games (e.g. Pong, Breakout). These are challenging reinforcement learning environments, as the state of the agent is represented in the form of high-dimensional RGB images, and the agent has to learn to extract relevant information from these images and play the game based on this. The authors of

this algorithm showed that this algorithm is capable of outperforming both other reinforcement learning methods and also human players in (at least some of) these games.

DQN is a model-free method, which means that the agent does not try to build an inner model of the environment based on its experiences, but takes actions based solely on the current observation. The algorithm is off-policy: it uses a separate target network, whose parameters are updated with the Q-network parameters only after taking some steps, which makes the training more stable.

DQN uses a deep neural network (e.g. convolutional neural network in many cases) to learn successful control policies from raw image data by extracting useful information from them. This network is trained with a variant of Q-learning to output a value function estimating future rewards. CNNs have proved to be successful in solving supervised image processing challenges, and they can be useful for finding similarities between and extracting important information from images. However, in the case of reinforcement learning, no labeled training data is available to train the neural network. Another difficulty that arises is that during training the network, in the case of supervised learning, the data samples are supposed to be independent and identically distributed; while in the case of reinforcement learning, as the agent plays the game, there is a strong correlation among consecutive images, and the data distribution can change significantly as the agent progresses in the game. To overcome these problems, DQN uses an experience replay buffer that stores the previous observations, actions, and rewards. The neural network is trained with gradient descent with samples from the replay buffer.

The goal of the agent is to choose the best action in each step that maximizes future rewards. The optimal action-value function $Q^*(s, a)$ is defined as the maximum expected return achievable when seeing state s and taking action a, following a policy π mapping observations to actions:

$$Q^*(s, a) = \max_{\pi} \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right] \tag{2.3}$$

. The optimal action-value function satisfies the Bellman equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a') | s, a\right] \tag{2.4}$$

. The action-value function is estimated by a function approximator, for example, a neural network with weights θ (also referred to as Q-network): $Q(s, a; \theta) \approx Q^*(s, a)$.

The Q-network can be trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right], \tag{2.5}$$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right]$ is the target for iteration i and $\rho(s,a)$ is a probability distribution over sequences s and actions a that can be referred to as *behavior distribution*.

2.3 Sim-to-real Transfer

Reinforcement learning has been successfully used to solve different kinds of challenges, for example, board games or computer games. However, using this technique for real-world robotic or autonomous driving applications is a much more difficult problem. The desired approach is to create a simulator that models the real-world machine, and use it to train an agent with RL to learn to do some kind of task (e.g. grasp an object with a robotic hand, or perform lane following in the case of an autonomous vehicle).

Using a simulator for training the agent has many advantages. Firstly, reinforcement learning relies heavily on trials and errors and uses the experience gained from previous failures and successes to learn an optimal control policy. However, running a trial-and-error method on a real robot can be dangerous or expensive. For example, we cannot allow a robot to hit objects or an autonomous vehicle to crash into other objects. Learning to avoid such scenarios is much safer in a simulator than in the real world. Secondly, training neural networks requires a huge amount of (labeled) data, which can be expensive and time-consuming to collect in the real world. Using a simulator, it is possible to collect even an unlimited amount of labeled and diverse data quickly. Also, with sufficient computing resources, it is possible to train an agent in the simulator faster than in real-time. 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.

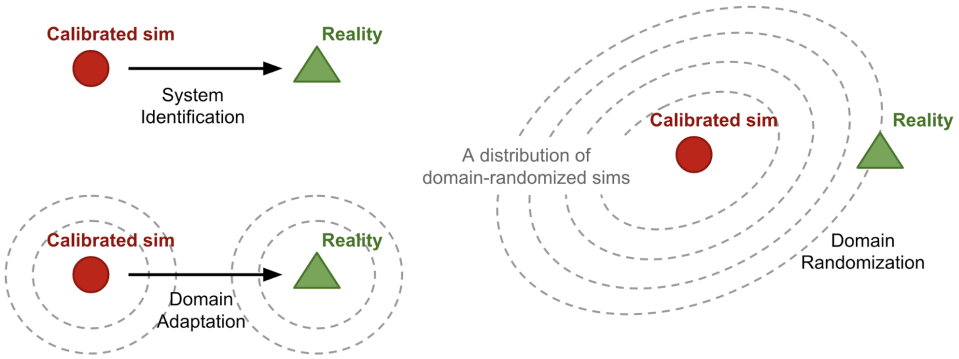


Figure 2.3: Overview of the approaches of sim-to-real transformation (source: [19]).

However, training agents in simulator introduces new challenges. Firstly, we need to find or create a simulator that models the world as accurately as possible, which can 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 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 2.3.

2.3.1 Domain Adaptation

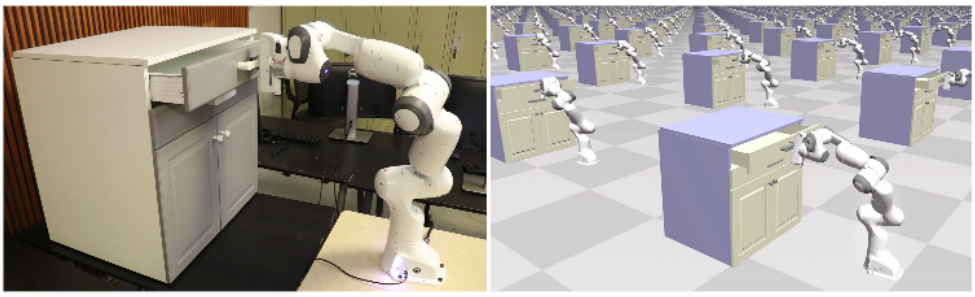
Domain adaptation is used in a much broader spectrum of machine learning applications than just robotics and reinforcement learning. When training neural networks (or other machine learning models), we often assume that the training data has a similar distribution to the data the model will be used on. 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.

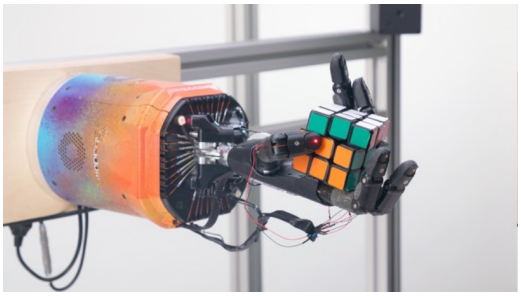
Domain adaptation can be imagined as when someone knows how to speak Italian, and would like to learn to speak Spanish. Using his/her previous Italian knowledge can significantly help to learn the new language. There are many examples in machine learning, where the source and target domain of training 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.

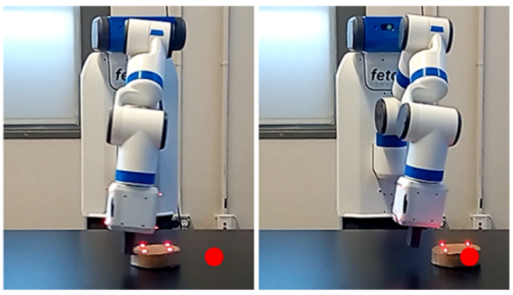
One approach to cope with the shift between the domains is instance reweighting. Samples from the source domain that are closer to the distribution of the target domain are considered with higher weights, as they can be used more accurately for the predictions from the target domain; and other samples are considered with lower weights. [20] 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 [21]. 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.



(a) Opening a drawer with a robot hand (source: [25]).



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



(c) Pushing objects to a certain location (source: [26]).

Figure 2.4: Domain randomization can be used to solve many kind of robotic tasks.

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 transformation, their distributions become more similar. [22] shows an example of this technique, which they apply to image classifiers trained on one dataset to recognize the instances of other datasets. [23] 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. [24] 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

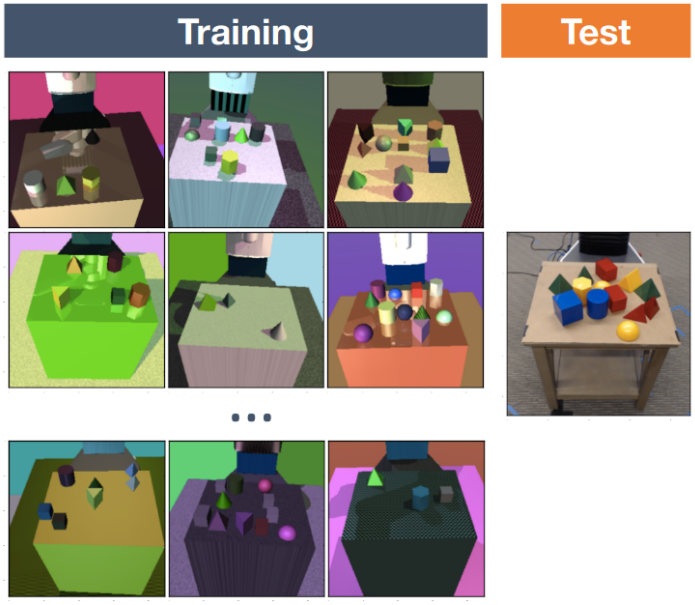


Figure 2.5: Illustration of domain randomization for robotic grasping (source: [27]).

the environment with different parameters. Training an agent that works across all randomized environments can be expected to work in the real world as well if the real-world environment 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 2.4.

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:

$$\theta^* = \arg\max_\theta \mathbb{E}_{\xi\sim\Xi}\left[\mathbb{E}_{\pi_\theta,\tau\sim e_\xi}\left[\mathcal{R}(\tau)\right]\right] \tag{2.6}$$

where τ_ξ is a trajectory collected in source domain randomized with ξ ([19]).

[27] successfully uses domain randomization to train a neural network in a simulator which 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 2.5. 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. [28] 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, they render synthetic objects (e.g. cars) 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 the real-world images; 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 [26]. 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 can be found in [25]. 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 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

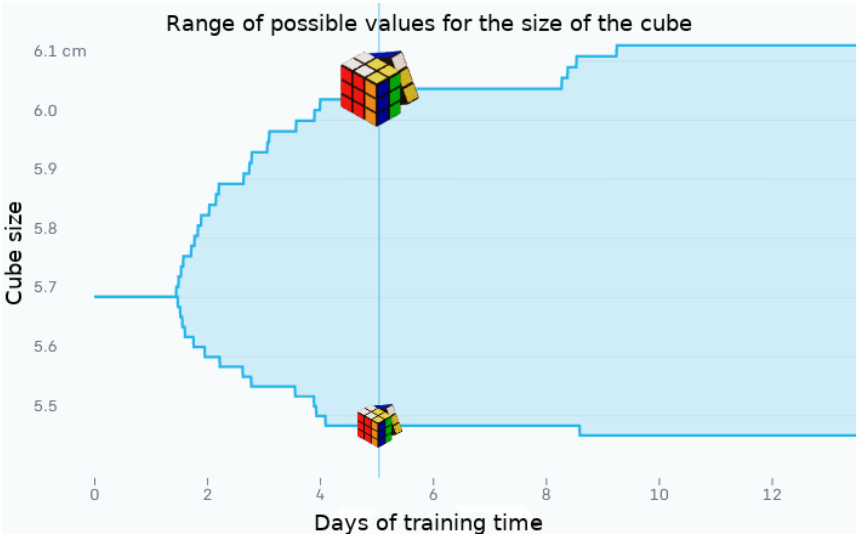


Figure 2.6: Automatic Domain Randomization automatically widens the randomization parameters during training (modified figure of [29]).

successful policy transfer for two robotic tasks: opening a drawer with a robot hand (see Figure 2.4a) 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, 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 Automatic Domain Randomization are the following. Training in the environment without domain randomization results in a policy that works well in the environments, but may overfit to the specific parameters of the simulator, and thus will not work on the real robot. 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, Automatic Domain Randomization 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 during training (see

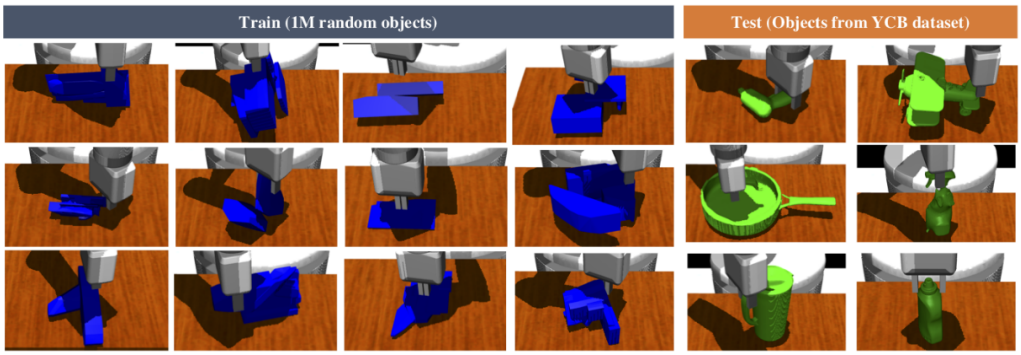


Figure 2.7: Training a model on lots of procedurally generated objects makes it able to generalize to real-world objects as well (source: [30]).

Figure 2.6). 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 Automatic Domain Randomization 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 Automatic Domain Randomization performs better than "traditional" domain randomization with fixed randomization ranges.

[30] 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 [31]. This model has to be built based on observing 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.).

[32] 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 Previous work

Previous results of my research are presented in [33]. In this paper, we (me and my supervisors) used a different method for training the agent and transferring it to the real-world vehicle. The current report 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.

Chapter 3

Proposed method

In this section, I describe the proposed method. An overview of the method is shown in Figure 3.1. In each timestep, the environment provides the observation S_t in the form of a raw RGB image to the agent. These images go through several preprocessing steps to form an image sequence S'_t. The agent receives the image sequence and calculates an action A_t. During inference, the actions are post-processed (A'_t) to make the movement of the robot smoother. The environment calculates the next state of the agent and provides a reward function R_t that describes how good the agent's action was. The agent uses the value of R_t to learn a good policy.

I describe each part of the proposed method in the following sections.

3.1 Environment and simulation-to-real transfer

To train the agent an autonomous driving simulator is used. My goal is to train an agent in such a way that it successfully performs lane following based on visual input both in the simulated and the real-world environment. I can achieve this goal without any further training on real-world data. I.e. the agent is only trained in the simulator, and the trained model is able to control the vehicle both in the simulated and the real-world environment.

Due to the differences between the simulated and real-world environments, agents trained in the simulator tend to perform poorly in the real world. In the training phase, I randomize several parameters of the simulated environment. This way the agent learns to navigate in several different environments, which helps the models to generalize better without overfitting to the specific properties of the simulator.

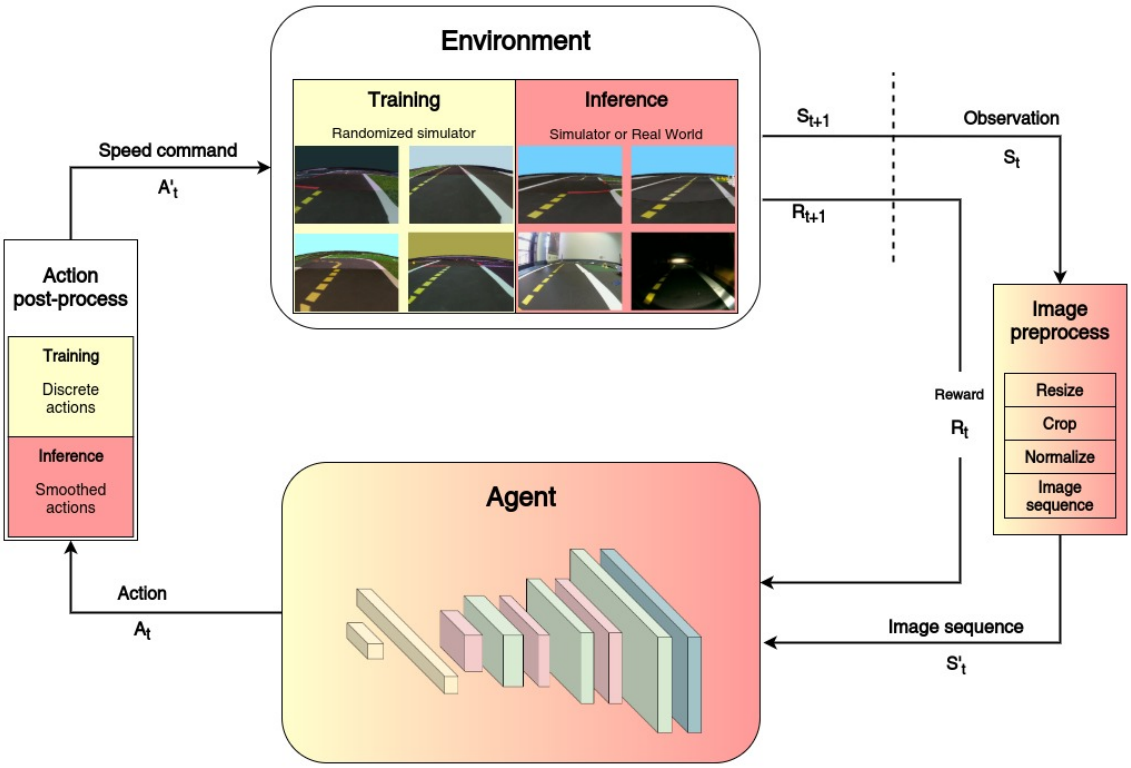


Figure 3.1: Overview of the proposed method. In each timestep, the environment provides the state to the agent in the form of high-dimensional raw RGB images. During training, several parameters of the simulator are randomized for a successful sim-to-real transfer. The images go through several preprocessing steps, and thus an image sequence is formed, which is received by the agent. The agent uses a CNN to predict an action command. At inference time the commands are smoothed for more sophisticated navigation.

Also, the agent is able to perform in the real-world without severe performance degradation. My method belongs to the broad family of domain randomization.

More formally, during training, I have a set of M simulator parameters whose values I can change. These parameters include both physical and visual parameters.

Physical parameters are the speed of the robot, camera position, camera field of view, and the size of the robot in different dimensions. For each parameter λ_i, I have a randomization range $[P_i, Q_i]$. At the start of each episode, 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 are sampled independently from each other and independently from their values in the previous episodes. These parameters define

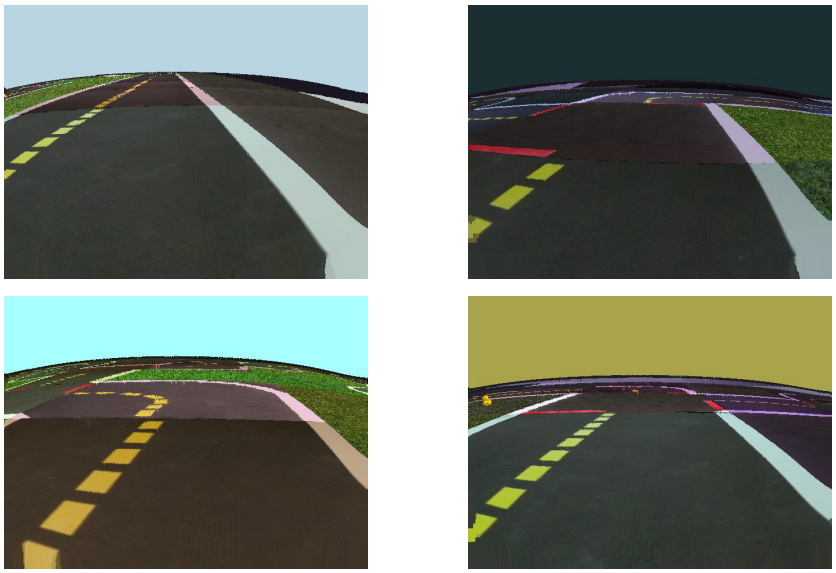


Figure 3.2: During training, the visual parameters, such as colors and textures, of the simulation environment are randomized.

a new environment in each episode. The list of the randomization parameters and their randomization ranges can be found in Appendix A.1.

Visual parameters include the colors and textures of the objects and global lighting conditions. Figure 3.2 shows some examples of the effects of randomizing visual parameters.

3.2 Image preprocessing

The environment (either it is the simulator or the camera of the real-world vehicle) provides the state S_t in a form of a raw high-dimensional RGB image. I use four steps to preprocess these images to make it easier for the agent to learn a good policy. The same preprocessing steps are used both during training and inference. These preprocessing steps are:

1. **Resize**: The images are resized from their original resolution (e.g. 640×480) to a smaller size (e.g. 80×60). The advantage of this step is that smaller images can be processed faster by the CNN, thus, decreasing both training and inference times (note that the latter is critical to realize real-time vehicle control).

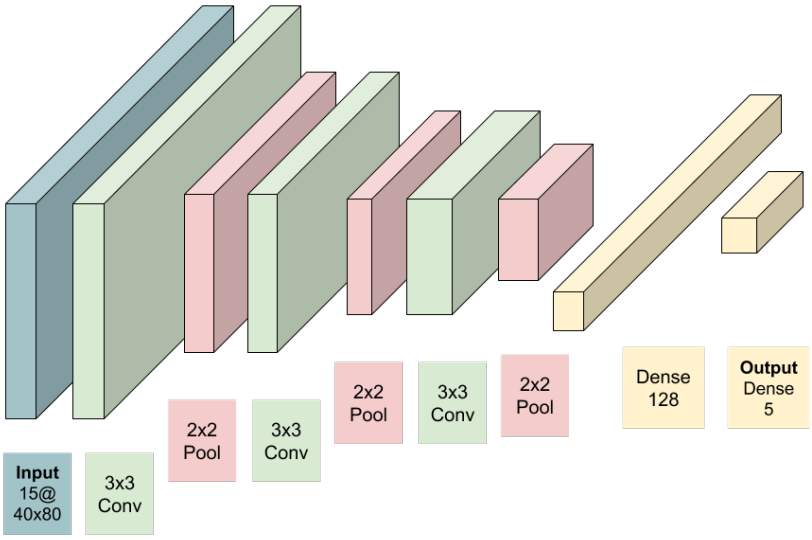


Figure 3.3: The architecture of my policy network. It had to be designed carefully such that the images can be processed in real time in order to successfully control the vehicle.

2. **Crop**: The upper part of the image is cropped because it does not contain useful information for lane following (cropping is done typically above the horizon). This step also reduces the size of the images, thus making the length of the training of the neural network and inference time smaller.

3. **Normalize**: The values of the image's pixels are rescaled from their original $[0, 255]$ range to $[0, 1]$. This step makes the training of the neural networks more stable, as the weights and the data are on a similar scale.

4. **Image sequence**: Instead of using the latest image $[S_t]$ as the state, a sequence of k images form the actual state S'_t. Thus, the state of the agent is described by $[S_{t-k+1}, ..., S_t]$ in a form of a tensor with the shape $image\_height \times image\_width \times 3k$. This describes the state of the agent more accurately than one single image instance.

The resulting image sequence S'_t is used as input for the agent's policy network.

3.3 Training the agent

I use the DQN algorithm to train a policy which can control the vehicle. I train a convolutional neural network with the preprocessed images. The input of the network is an image sequence S'_t width the shape $image\_height \times image\_width \times 3 \cdot sequence\_length$. For fast inference I utilize a simple neural network; it is critical to process the images in as little time as possible to successfully control the vehicle. My neural network can process the images in a few milliseconds on a computer with limited hardware resources, which is suitable for real-time control.

The architecture of the applied neural network for learning the policy can be seen in Figure 3.3. It consists of three convolutional layers, each followed by ReLU (non-linearity) and MaxPool (dimension reduction) operations. The convolutional layers use 3×3 kernels with $32, 32$ and 64 filters. The MaxPool layers use filters with a size of 2×2. The last pooling layer is followed by two fully connected layers with 128 and 5 outputs. The output of the last layer defines the probability distribution predicted for the actions.

3.3.1 Reward function

The goal of a good reward function is to describe how accurately the vehicle follows the lane. When the agent goes well in the right lane, it 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).

I used the following formula as the reward function. When the robot is on the road, the reward is calculated as:

$$r = \alpha_v \cdot v \cdot \cos\phi + \alpha_d \cdot d + \alpha_p \cdot p_c, \tag{3.1}$$

where v is the speed of the vehicle in the simulator, ϕ is the angle between the heading of the vehicle and the tangent of the optimal curve, 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 $\alpha_v \cdot v \cdot \cos\phi$ 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

Table 3.1: The discrete actions predicted by the agent are mapped to wheel speeds. (Maximum speed is 1.0.)

Action	Left wheel speed	Right wheel speed
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

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.

When the vehicle is in an invalid position (e.g. it leaves the track), it gets a penalty of $-p_x$. The simulated episode also ends at this point.

3.4 Action post-processing

The agent predicts one of the 5 possible actions in each timestep. The actions are mapped to wheel speed commands according to Table 3.1.

During training, the agent tries to learn a policy to choose the best action sequence in an episode. However, during inference, the usage of discrete actions results in a crude movement of the vehicle. To make it more sophisticated, instead of selecting the action predicted with the highest probability, I use A'_t which is a combination of all actions. I use the probability distribution the agent predicted for the actions and calculate its inner product with the wheel speeds of each action. Formally, if we denote the probability distribution predicted by the agent for the actions by $p_1, \ldots, p_N$ (where N is the number of possible actions) and the actions' wheel speeds by $[L_1, R_1], \ldots, [L_N, R_N]$, then the final wheel speed commands are calculated as $[L_{final}, R_{final}] = \sum_{i=1}^{N} [p_i \cdot L_i, p_i \cdot R_i]$. My experiments showed that this method results in a much smoother vehicle control than using discrete actions.

Chapter 4

Environment and implementation

In this section, I introduce the simulated and the real-world Duckietown environment which I used in my work, and I present how I applied the proposed method in this environment.

4.1 Autonomous driving environments

Training self-driving vehicles in a simulator rather than on the real car 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 many self-driving car simulators that can be used to train models for autonomous driving. Some of them are more complex, while some of them are simpler. Selecting a good simulator that fits the needs of my current work is an important step. As the focus of this report 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 provides a reward function which represents the goodness 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 less power-demanding makes it possible to run several experiments within a reasonable time and compare their results.

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

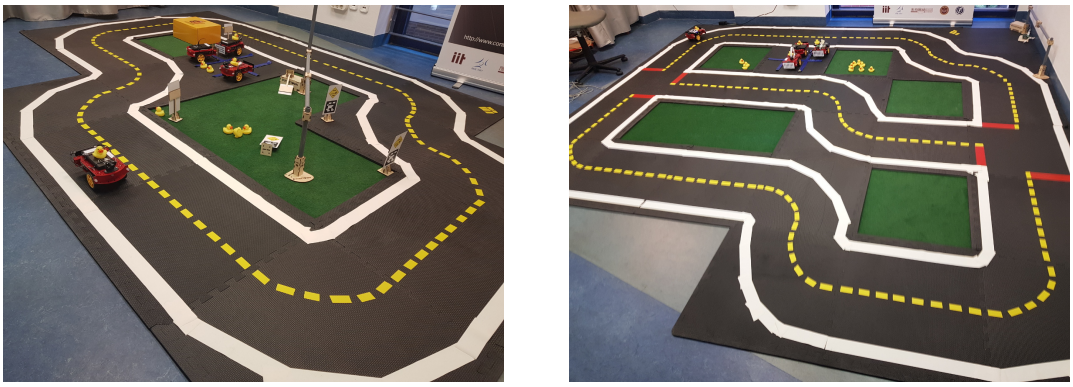
Duckietown[1] [9] is an educational and research platform where low cost robots (*Duckiebots*) travel in small cities (*Duckietowns*). The platform is made of inexpensive, off-the-shelf elements, which makes it easily accessible and ideal for education and research. The environment is highly flexible: using standardized elements, different kinds of cities can be built. The platform offers a wide range of functionalities and challenges: it can be used for research related to robotics, embedded artificial intelligence, machine learning, and autonomous driving, or 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 robots which 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[2] [34] are a series of competitions on the Duckietown platform organized at the recent NeurIPS and ICRA conferences. So far, three rounds of competitions have been held; the fourth 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 different difficulties: for example, in the last round of the AI-DO, these challenges were 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 robots), 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 robots, 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

[1] https://www.duckietown.org/ Access date: 27 October 2020.

[2] https://www.duckietown.org/research/ai-driving-olympics Access date: 27 October 2020.



(a) (b)

Figure 4.1: Example setups of the Duckietown environment in the AI Showroom of BME TMIT: (a) shows a simpler map with a single road, while (b) contains intersections.

of the proposed 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 4.1 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 of maps can be built. Simpler maps contain only a single loop (see Figure 4.1a), while more complex setups can also contain intersections as well (see Figure 4.1b).

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 robot has to navigate in the right lane along the road. Besides, more complex challenges are also possible: for example, when the map includes intersections, handling them can also mean difficulties. When more robots are navigating on the map at the same time, avoiding collisions is also an important task that can be solved. The platform is also suitable for more complex tasks, for example, path planning and cooperation between the vehicles.

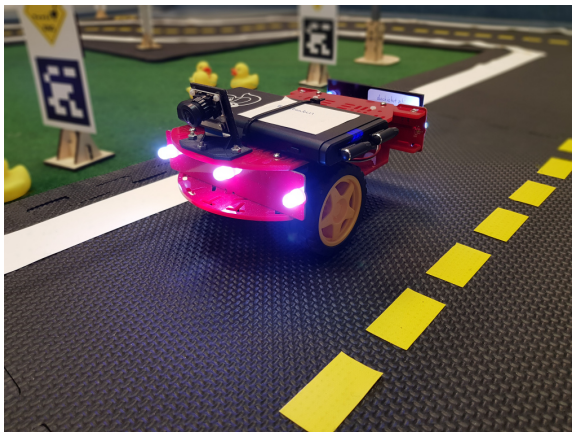


Figure 4.2: Duckiebots are small three-wheeled robots with a forward-facing camera. It has to be controlled by processing the camera images and giving wheel speed commands.

4.2.2 Duckiebots

Duckiebots are vehicles that can be driven around the Duckietowns. The Duckiebot can be seen in Figure 4.2. These are small three-wheeled robots built of inexpensive, standard parts. The "brain" of the robot is a Raspberry Pi 3 minicomputer, which can be used to control the robot. The robot is powered by two DC motors that drive the two wheels of the robot; 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 robot is a forward-facing, wide-angle monocular camera, whose image has to be processed to navigate the vehicle.

4.2.3 Duckietown simulation environment

The Duckietown platform includes a software library[3] which includes the necessary tools to interact with the infrastructure and the robots. This includes components for communicating with the robot and controlling it manually or automatically with algorithms; a simulator which 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 [35] is the part of the software library that I would like to present in more detail, as I use it to train my models. The simulator creates a

[3] https://github.com/duckietown/. Access date: 27 October 2020.

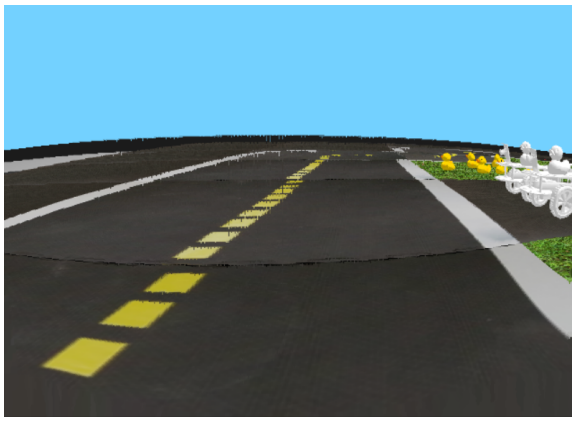


Figure 4.3: Camera image of the vehicle in the Duckietown simulator.

simulated view of the Duckietown environment: the virtual robot can be controlled with the same wheel speed commands as the real one, and the simulator generates similar images to the ones that the robot would see in the real world. An example of an image from the simulator can be seen in Figure 4.3.

The simulator is highly customizable. For example, we can create custom maps and use them in the simulator. Also, it can be used to train models to control the robot more effectively than training on the real robot. The simulator supports reinforcement learning by providing a reward function that describes how accurately the robot follows the right lane. When it is driving precisely in the lane, it receives high rewards; when it leaves the optimal curve, it starts receiving smaller rewards, and when it goes to the oncoming lane, it receives penalties. The robot also receives a high penalty when it leaves the track, and the simulated episode also ends at this point. This reward function can be used to train reinforcement learning agents: these agents try to maximize the earned reward, which can be reached by following the lane as precisely as possible.

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 possibilities of using reinforcement learning to train autonomous driving agents in a simulator and test the possibilities of transferring them to the real-world robot. A real-world Duckietown environment is assembled at the university, which can be used for testing agents on real robots. The Duckietown environment is simpler than most complex autonomous driving environments, which makes it possible to try and experiment with different methods and getting a response in a reasonable time; but it still provides scientific challenges which are, in terms of

difficulty, comparable to the ones in more complex environments. Thus I decided to work with this environment and use it in my experiments.

4.3 Implementation details

In this section, I present how I applied the proposed method in the Duckietown environment. I describe the details of training the agent, including the exact hyperparameters and the reward function I used.

4.3.1 Training hyperparameters

I used the DQN implementation available in the Stable Baselines collection [36] and the Tensorflow software library. I used the Weights & Biases Experiment Tracker [37] to keep track of my experiments.

The architecture of my CNN policy network can be seen in Figure 3.3. The input of the neural network S'_t is an image sequence of the last 5 images, which is a tensor of shape $40 \times 80 \times 15$. This image sequence is the result of resizing the images from their original size 480×640 to a smaller size 60×80, and then cropping the upper one-third of them. These smaller images still contain enough information for navigation. In each timestep the algorithm chooses one of the five possible actions; the mapping of actions to wheel speed commands is described in Table 3.1.

I did a manual hyperparameter optimization by experimenting with different parameter settings, including different values and settings for the learning rate, input image size, experience replay buffer size, discount value, policy network parameters, and wheel speeds. The parameters that gave the best results were the following. I used a batch size of 32, $\gamma = 0.99$ for the discount factor, the learning rate for the Adam optimizer was set to $5 \cdot 10^{-5}$, the size of the replay buffer was set to $150,000$, and the agent collected experience from $10,000$ steps of random actions before actually starting learning. I ran the training for $1,500,000$ timesteps, which took approximately 46 hours on an NVIDIA DGX Workstation, which contains 4 pieces of V100 GPUs, an Intel Xeon E5-2698 v4 2.2 GHz (20-Core) CPU, and 256GB of RAM.

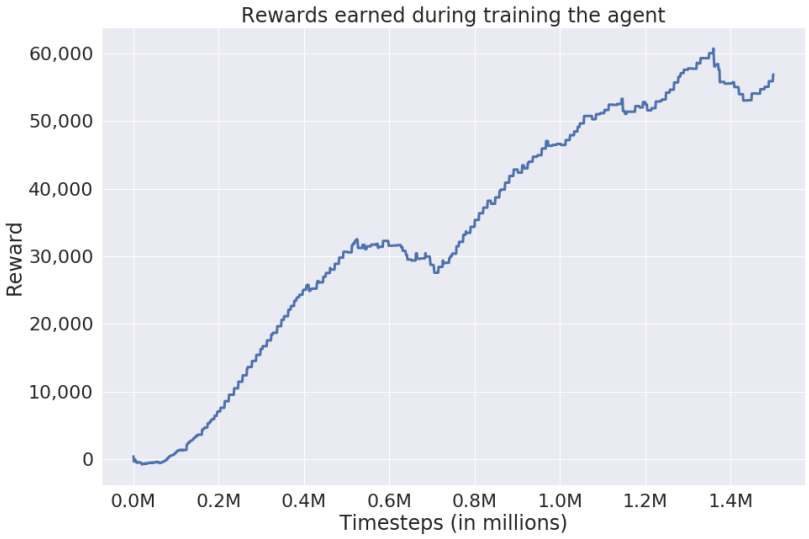


Figure 4.4: Reward earned by the agent in the training.

4.3.2 Reward function

The goal of a good reward function is to help the agent to learn to follow the optimal curve of the road as accurately as possible. The formula of the reward function I used, as described in more details in 3.3.1, is:

$$r = \alpha_v \cdot v \cdot \cos\phi + \alpha_d \cdot d + \alpha_p \cdot p_c.$$

When the vehicle leaves the lane, it receives a penalty of $-p_x$.

In my implementation I experimented with different values for the coefficients α_v, α_d and α_p. The values of these coefficients determine how significant each factor in the reward function is, thus it is important to set their values properly to create a meaningful reward function that encourages the agent to do the desired behavior. The values that gave me the best results were the following. I set $\alpha_v = 10$, $\alpha_d = -100$ and $\alpha_p = 400$. Note that α_d should have a negative value, as driving further from the center of the lane is discouraged. When the vehicle leaves the road, it gets a high penalty $-p_x$ (since we don't want this scenario to happen). The value of this should be set to be higher than the rewards received during driving; I observed that the latter were mostly between -10 and 10, so I used $p_x = 40$.

The rewards earned throughout the episodes of training can be seen in Figure 4.4.

Chapter 5

Evaluation and results

In this chapter, I will evaluate the proposed method and analyze its performance.

I tested my trained agent both in the simulator and the real world and compared its performance in the two environments. I also tested the performance of the agent in extreme test cases, which the agent was not taught explicitly to be able to handle. For example, it can navigate in night vision conditions, even though in the simulator it was trained only in daytime conditions. It is also able to recover to the right lane when it is started from an invalid starting location. A video presenting the test results can be seen at https://youtu.be/RiXQOt-mgZU.

5.1 Evaluation methodology

Finding a good evaluation methodology to analyze the performance of autonomous vehicles is a difficult problem due to the complexity of the task. The AI Driving Olympics platform provides metrics to measure the agent's performance, but these are designed for the competition. These prefer fast driving, while in my case, the goal is to create a reliably-performing agent. Thus, I developed the following evaluation methodology and used it to measure the performance of the proposed method.

The training and evaluation of the agent were carried out on different maps to avoid the possibility of overfitting to one single map. For training, I used a large map which contains a diverse set of road sections (e.g. long straight roads, S-turns) to prepare the agent to learn to drive in different scenarios. This map can be seen in Figure 5.1.

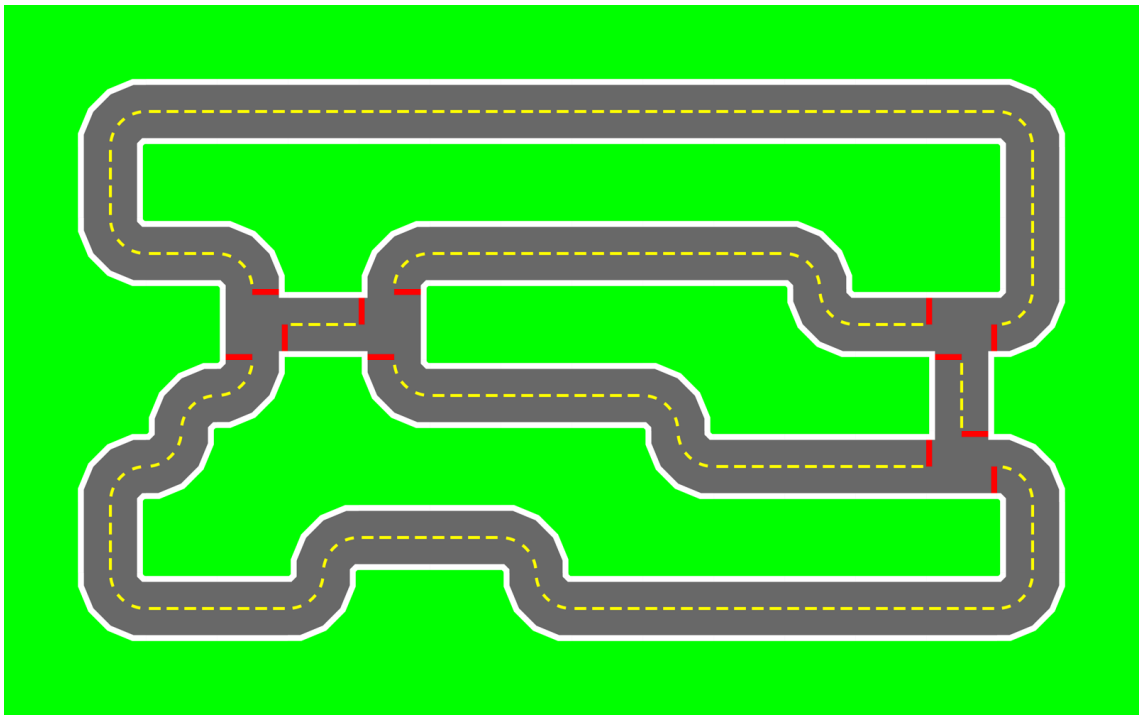


Figure 5.1: For training, a larger map was used, which enables the agent to learn to drive in various scenarios, e.g. long straight roads or S-turns.

I used different maps for testing the performance of the agent; the tests were carried out on them according to the following. On each test map, I ran several test cases. In each test case, the agent was started from a randomized starting location on the map (the precise ways of choosing randomized starting locations in the simulator and the real world are described in the corresponding sections). The duration of each test case was chosen based on the current test map such that the test case is long enough for the vehicle to drive at least a complete lap on the track. In this time, it has successfully passed all parts of the track, thus it can be assumed that it would be able to drive more laps as well. A test case was considered successful if the agent was able to drive the vehicle without leaving the right lane during the whole test. When the map included intersections, the agent was allowed to choose any direction at these points.

The performance of the agent was measured by the ratio of successful test cases. This evaluation methodology reflects the reliability of the tested agent.

5.2 Performance in the simulator

The performance of the agent in the simulator was tested on four different maps. One of them, Map #1, has the same trail as the real-world map. The maps can be seen in Figure 5.2.

In each test case, the agent was started from a randomized location on the map. More precisely, I chose a random starting position and a starting angle; the location was considered valid if the starting position was on a road tile (i.e. not outside the road), and the angle between the vehicle's head and the tangent of the optimal curve at the vehicle's location was at most 20 degrees. (Note that it was not a requirement that the robot must head towards the direction of travel in its lane; i.e. in some cases, it was started from the oncoming lane, and first had to navigate to the right lane.) I excluded those test cases where the agent was not able to drive at least 50 timesteps (this happened when it was started from the side of the road facing outwards, from where it is impossible to navigate back to the road). The duration of each test case was 2500 timesteps on Maps #1 and #2, 3000 timesteps on Map #3, and 6000 timesteps on Map #4; these are enough to drive a complete lap on every test map.

The results of the tests can be seen in Table 5.1.

I also ran longer tests in the simulator to verify that the agent can drive the vehicle on longer distances as well. These test cases lasted 50,000 timesteps and they were done on Simulator Map #1 (more than 20 complete laps) and Simulator Map #4 (more than 8 complete laps). Three tests were carried out on both maps. The agent was able to drive the robot without leaving the right lane for the whole length of all tests, which certifies that it can control the vehicle on longer distances as well.

Table 5.1: Rates of successful drives on four simulated and one real-world map.

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	28	93.3%
Real world	30	27	90%

5.3 Performance in the real world

The setup of the real-world Duckietown environment can be seen in Figure 4.1b. For the tests in the real-world environment, the procedure was similar to the one in the simulator. In each test case, the vehicle was started from a randomly selected position on the map, heading roughly towards the correct direction (a few degrees of difference was allowed). Each test case lasted 50 seconds, which is enough for the robot to take a complete lap. The success rate of the tests can be seen in Table 5.1.

In order to successfully control the vehicle in the real-world environment, it is essential to use algorithms that can be run in real-time with as little latency as possible. My neural network (see Figure 3.3), which processes the images, was designed such that it can be run efficiently on a computer with limited hardware resources (for testing I used a laptop with no dedicated graphics card and an Intel®Core™i7-4500U CPU @ 1.80GHz). According to my measurements, the total time required to process an image (i.e. preprocessing steps and inference on the neural network) was 5-10 milliseconds, which is sufficient for real-time vehicle control with the camera images arriving at a rate of 25-30 FPS.

To emphasize the importance of using domain randomization techniques, I ran a training where I used none of the simulator randomization techniques described in Section 3.1, but the other parameters of the training were unchanged. The results of this training were the following. This model was able to drive the agent in the simulator just as well as the one trained with randomization, however, when transferred to the real-world vehicle, it was unable to drive more than 5 tiles on the road, thus being really far from driving a complete lap. This shows the importance of using the randomization techniques I described in Section 3.1.

5.4 Agent navigation patterns

To understand the behavior of the agent better, I created several figures showing the paths of the vehicle both in the simulator and in the real-world environment.

5.4.1 Simulator

In the simulator, the position of the robot was drawn on the map in each timestep. These maps can be seen in Figure 5.2. The initial location of the vehicle is marked

with a red circle. Most maps show successful test cases where the agent was able to follow the middle of the lane. In some cases, for example, in 5.2c and 5.2f, the agent was started from the oncoming lane, but it was able to navigate back to the right lane. Figs. 5.2d and 5.2h show examples of two failed test cases, where the agent was not able to follow the right lane. In these cases, the vehicle was started from the oncoming lane at the corner of the track, from where it is really difficult to find the right lane. Some uncertainty can be noticed in the behavior of the agent in Figure 5.2a at the intersection; this can be the result of not specifying which direction the agent should follow, thus it "decides" it only in the middle of the intersection. However, the agent was able to drive a complete lap without leaving the right lane in these cases too.

5.4.2 Real world

Tracking the path of the vehicle in the real world is a more challenging problem compared to doing the same in the simulator. I used a wide-angle camera placed in a fixed position above the track to record the movement of the robot. Then I followed the path of the vehicle on the recording using the Discriminative Correlation Filter Tracker with Channel and Spatial Reliability (CSR-DCF) algorithm ([38]). I used the implementation of this algorithm which is available in the OpenCV software library [39]. I plotted the position of the center of the vehicle from each frame of the video onto a single image to visualize its path. The result can be seen in Figure 5.3.

Due to the distortion of the wide-angle camera, the plotted path (especially on the right side of the image) is not completely accurate. This part of the track is recorded by the camera from a higher angle, thus the projection of the top of the robot to the track does not fall into the position of the bottom of the robot; instead, it is projected towards the middle lane, thus the line in the right part of the figure seems to be on the middle lane.

5.5 Extreme test scenarios

To justify the robustness of the proposed method, I tested the agent under extreme test conditions. These are mostly situations that the agent was not trained explicitly to be able to handle. Based on my experiments, the agent was able to cope with

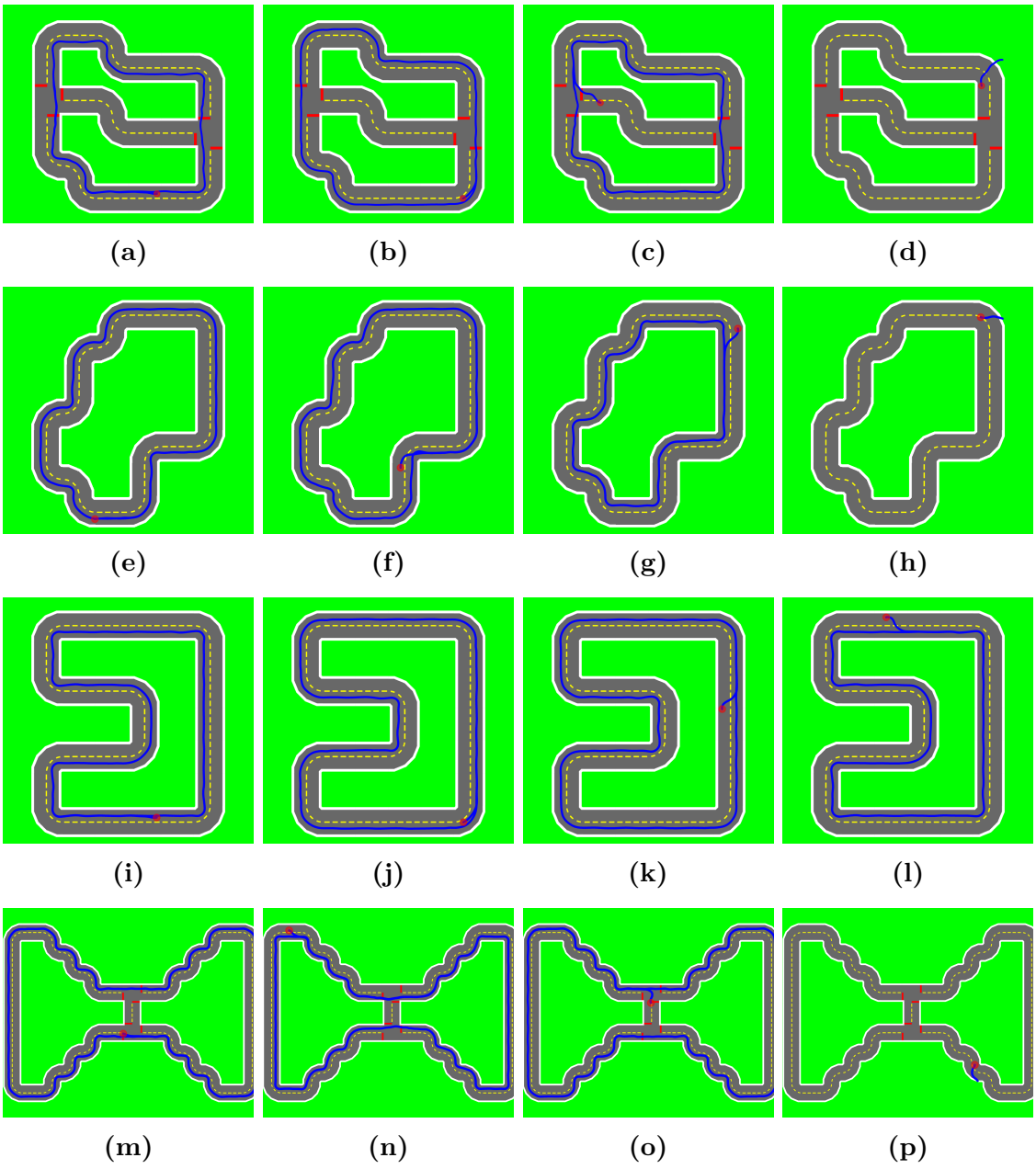


Figure 5.2: Agent navigation patterns in the simulator. Figures (a)-(d), (e)-(h), (i)-(l) and (m)-(p) show the paths of the vehicle in Simulator Map #1, Map #2, Map #3, and Map #4, respectively. The starting location of the vehicle is marked with a red circle, and its path is drawn with a blue line. While most pictures show examples of successful test cases, figures (d), (h), and (p) show examples of failed test cases. In some cases (e.g. (c), (f)) the robot was started from the oncoming lane, but it was still able to navigate back to the right lane and drive a whole lap there.

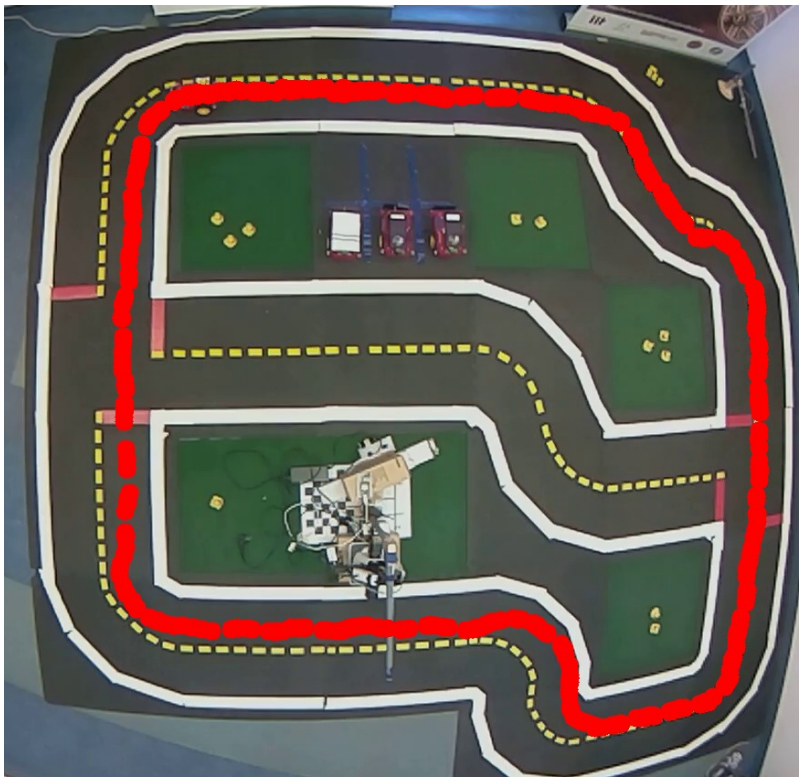


Figure 5.3: The path of the agent in the real-world environment. The picture was created by recording a video with a camera placed above the track and using the CSR-DCF object tracking algorithm [38]. Due to the camera not being completely above the track, the image has distortion on its right side, and the path is not displayed accurately here.

these extreme test conditions. The results of the extreme test cases are summarized in Table 5.2. All of these tests were carried out in the real-world environment.

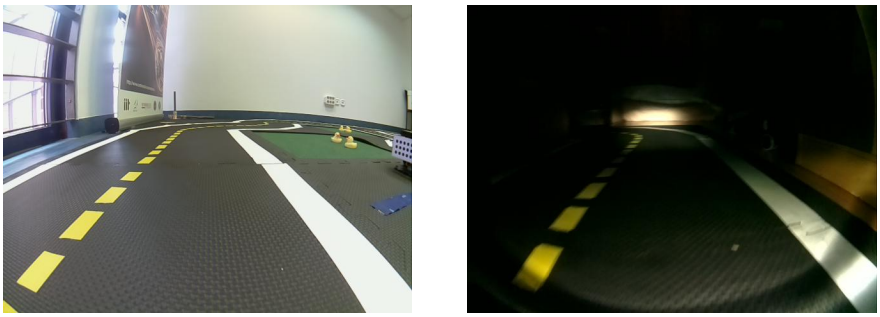
5.5.1 Night mode

The daytime and night vision conditions provide significantly different observations for the vehicle. Despite being taught only in daytime lightning conditions in the simulator, the agent can navigate the vehicle in the real-world environment in night visual conditions.

A comparison of the daytime and night visual conditions can be seen in Figure 5.4. Notice that the camera images look significantly different under daytime and night visual conditions. The tests in night visual conditions were carried out in a completely dark room. A flashlight was installed on the robot to imitate the

Table 5.2: Success rates of the extreme test cases.

Extreme test case	Test cases		
	Total	Successful	Success rate
Night visual conditions	5	4	80%
Starting from oncoming lane	3	3	100%
Starting crossing the road	4	4	100%
Driving slower	3	3	100%
Driving faster	3	3	100%



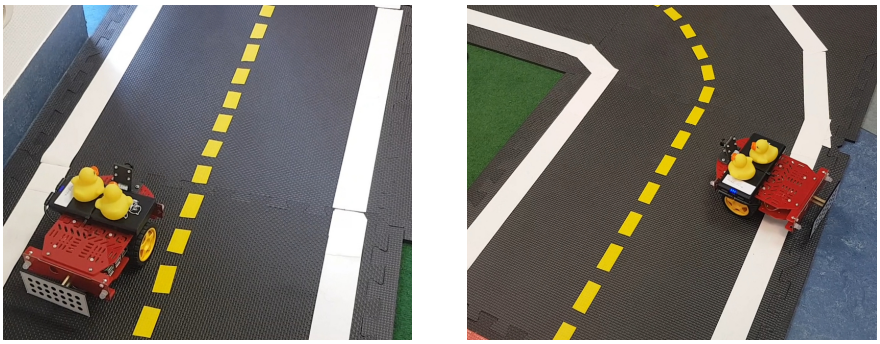
(a) **(b)**

Figure 5.4: Camera images from the robot in daytime (a) and night (b) visual conditions. The agent was taught in the simulator only in daytime visual conditions, but it is able to drive in night visual conditions as well.

headlights of a vehicle. Due to technical difficulties, the night mode tests were carried out on a smaller track compared to the one I used for the regular tests. The testing methodology was similar to the one described in 5.1. Under night visual conditions I ran five tests; in four of them, the agent was able to drive for a complete lap on the track. A video of the agent driving in night visual conditions is included in https://youtu.be/RiXQOt-mgZU.

5.5.2 Recovery from invalid starting position

My experiments show that the agent can recover from invalid starting positions. Invalid starting positions include starting from the oncoming lane or cross the road; see Figure 5.5. The agent was never trained explicitly to be able to do so; it could only use the values of the reward function and trial-and-error experiments to learn to handle such scenarios.



(a) **(b)**

Figure 5.5: Extreme test cases include starting the vehicle from invalid locations: the oncoming lane (a) and cross the road (b). My experiments showed that the agent is able to recover from these starting positions to the right lane and continue driving there.

In these experiments, the vehicle was started from invalid locations in different positions of the track. Each test case lasted 10 seconds, which is enough to recover from the invalid location to the right lane. If the agent was able to find the right lane and start driving there, the test case was considered successful. The results can be seen in Table 5.2: the agent was able to recover in all test cases. The recovery can also be seen in the video https://youtu.be/RiXQOt-mgZU.

5.5.3 Modifying the vehicle speed

The third type of extreme test cases included modifying the speed of the vehicle. This was carried out by multiplying the original speed values (see Table 3.1) by a constant value. In contrast to the previous extreme tests, the agent was taught to be able to drive in the simulator at different speeds. However, driving faster or slower is still not an easy task: for example, when the image sequence is formed of the last five camera images, different speeds mean that the time difference between consecutive images changes.

I made two kinds of tests: in the first ones, the vehicle's speed was multiplied by a constant factor of 0.8 for slower movement, and in the second ones, it was multiplied by 1.2 for faster movement. My experiments showed that the agent was able to drive in both conditions; see Table 5.2 for the details.

Chapter 6

Summary

Training real-world autonomous driving agents using deep reinforcement learning is a challenging task in several aspects. The desired approach is to use an autonomous driving simulator where the agent can carry out trial-and-error experiments without high costs and risk of accidents. Firstly, training an autonomous driving agent in a simulator with deep reinforcement learning is already a challenging task, as these methods tend to be unstable and lack a proper mathematical basis. Secondly, when these agents are transferred to the real world without using proper transfer learning methods, they tend to suffer severe performance degradation.

In this work, I presented a method for training autonomous driving agents in a simulated environment and transferring them to real-world vehicles. I used the Deep Q-Networks algorithm to train the agent in an autonomous driving simulator. By using proper domain randomization techniques, I showed that the agent can successfully be transferred to the real vehicle without further training on any real-world data. I implemented and evaluated my method in the Duckietown environment, where the agent can successfully perform lane following based on camera input using my method.

I investigated the robustness of my method with extreme test cases that the agent was not taught explicitly to be able to handle. Using my method, the agent can control the vehicle in night visual conditions despite being trained only in daytime conditions, and it can successfully recover from illegal starting positions.

6.1 Future work

The topics of autonomous driving and sim-to-real transformation are popular and important research areas. I would like to publish my current method and results in a journal paper, and continue the research in this area afterwards.

As part of the future work, I would like to participate in the AI Driving Olympics and compare my method to the algorithms used by other researchers. To do this, I will optimize my method to fit the rules of the competition, where the goal is to drive as long distances as possible in a given time without leaving the track. This requires to specialize my method to be able to control the vehicle with higher speed, which has not been an aspect of the current work.

Possible future research can include a deeper analysis of the agent, to better understand its decisions and behavior. As deep neural networks are often viewed as "black box" elements, this is a challenging task, but a deeper understanding of the decisions of the agent is required for more complex real-world applications.

As an addition to the current lane following task, the method can be extended with obstacle detection and collision avoidance. This is also an important task in the development of autonomous vehicles. Also, the AI Driving Olympics competition features challenges of collision avoidance, which serves as a good opportunity to test these methods.

Acknowledgements

The author is pleased to thank his supervisor, Bálint Gyires-Tóth, for his continuous support and advice, which greatly contributed to the creation of this report.

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 research has also been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16- 2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

Bibliography

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[2] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[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(7676):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, Wojciech M Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, et al. Alphastar: Mastering the real-time strategy game starcraft ii. *DeepMind blog*, page 2, 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 *ICML*, 2010.

[11] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[12] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86 (11):2278–2324, 1998.

[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] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[15] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[16] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris

Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[18] 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, 2015.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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, 2006.

[24] 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.

[25] 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.

[26] 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 1–8. IEEE, 2018.

[27] 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. IEEE, 2017.

[28] 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.

[29] openai2019rubikblog. Solving rubik's cube with a robot hand, Oct 2019. URL https://openai.com/blog/solving-rubiks-cube/.

[30] 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.

[31] Lennart Ljung. System identification. *Wiley Encyclopedia of Electrical and Electronics Engineering*, pages 1–19, 1999.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] Lukas Biewald. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

[38] Alan Lukezic, Tomas Vojir, Luka Čehovin Zajc, Jiri Matas, and Matej Kristan. Discriminative correlation filter with channel and spatial reliability. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6309–6318, 2017.

[39] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

Appendix

A.1 Randomization ranges

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