From Simulated to Real Test Environments for Self-Driving Cars

Master's Thesis submitted to the

Faculty of Informatics of the *Università della Svizzera Italiana*

in partial fulfillment of the requirements for the degree of

Master of Science in Informatics

presented by

Brian Pulfer

under the supervision of

Prof. Paolo Tonella

co-supervised by

Dr. Andrea Stocco

June 2021

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Brian Pulfer
Lugano, 15 June 2021

Abstract

Self-Driving Cars (SDC) are currently one of the most ambitious applications of Artificial Intelligence and Engineering. To date, state-of-the-art techniques for building a SDC are based on the recent advances in Deep Learning, where the actions executed by the autonomous vehicle are predicted by one or more artificial neural networks.

As the industry is moving rapidly towards building SDCs, testing such systems is becoming more and more relevant. We consider two types of testing: *offline testing* where just the decision-making component of the car is tested and *online testing* where the whole system is tested. Online testing of real-world SDCs is both expensive and dangerous, while offline testing cannot give insight as to whether the system as a whole works correctly. To face this problem, online testing is also done on simulators which are cheap and safe but have the drawback of not being representative enough of the real-world inputs that the SDC might observe.

In this thesis, we compare online and offline testing of SDCs both in a simulator and the real world through the use of DONKEY CAR, a small-scale SDC open-source project. We realized a simulated testing environment that resembles our real testing environment and study which insights offline testing offers by considering its relation with online testing in both environments.

We ultimately find that physics-based simulators, despite being designed to match a real environment, are not representative of the real world from the perspective of machine learning models unless photo-realism is offered. We argue that online testing cannot be substituted by offline testing. This claim is supported by the results obtained in our experiments, where we found that models which did best in offline testing did not yield the best driving performances when tested in the online setting.

Acknowledgements

Throughout the writing of this thesis, made harder by the situation of pandemic caused by the COVID-19 virus, I was lucky to find support from a number of people.

Firstly, I would like to thank my supervisor, Prof. Paolo Tonella, and co-supervisor, Dr. Andrea Stocco, for their availability and support from the very first day I met them to the very last day of work with them at the Software Institute. Their supervision was key for realizing and directing this piece of work.

I am also thankful to all my classmates with whom I had the opportunity to learn much about artificial intelligence and other subjects during these two years of studies. I believe we shared not only knowledge and thoughts, but also personal growth.

Last, but definitively not least, I would like to thank all those people that have been near me outside the university and that I could always count on. They did an amazing job at making me happy. These include my supportive mother Mercedes, my inspiring grandmother Valentina, my beloved girlfriend Fabiana, my dear siblings Alain and Cristina, and my close friends Gionas, Jean-Jacques, Simone, Evgeny, Nicholas, Rosario, and more.

Contents

Figures

Tables

Chapter 1

Introduction

Autonomous Vehicles (AVs), and self-driving cars (SDCs), are vehicles capable of driving correctly and safely with little to no need for human control or supervision. The autopilot component used in such vehicles makes use of information about the surrounding environment gathered through a multitude of sensors to decide the next actions to be taken such as the speed at which the car should cruise, or the direction it should follow. Instances of sensors typically used in SDCs include cameras, radars, LIDARs (Light Detection And Ranging), GPS (Global Positioning System) [67].

SDCs are currently one of the most ambitious applications of Artificial Intelligence (AI) and Engineering, as the task of driving is known to be hard for any computer system. SDCs bring the promise of revolutionizing the way people use and perceive transportation [48].

First, SDCs are generally considered safer than human-driven cars. The processing and reaction times are much faster than those of human drivers and sensors provide more accurate and detailed spatial information about the driving scenario [66].

Second, SDCs are regarded as more accessible, since even people that do not meet the current requirements to obtain a driving license could benefit from car transportation in autonomy.

Furthermore, the advent of SDCs is expected to bring a number of collective advantages. One such advantage will be smoother traffic flow, as cars could come closer together in moments of traffic as they can sense close distances better than a human driver from its driving seat. With the advent of SDCs, there could also be fewer problems related to parking scarcity, as cars could drop people to their destination and autonomously park to a further place. SDCs are also expected to reduce the need of in-road signage and traffic police. Finally, passengers could use their time more fruitfully rather than focusing on the driving task [79].

In the last decade, breakthroughs in the field of AI have enabled the deployment of the first SDCs on public roads. Nowadays, the state of the art for self-driving car is the use of Deep Learning (DL), specifically Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) [72]. CNNs are widely used in image processing, as they can efficiently extract features and detect patterns. RNNs, on the other hand, allow to process temporal data, and are more context-aware.

Despite the advantages they bring in vision-based tasks, Neural Networks are notoriously hard to test. To date, it is also unclear to which degree DL system are capable of generalizing [27] since it is still hard to interpret the decisions they make [13]. Moreover, they are susceptible to adversarial attacks even without being directly modifiable by the attacker [61].

This makes the current SDCs still less safe than human drivers in many situations. Indeed, the number of possible scenarios that the car must handle safely is still enormous, and current DL systems need to be continuously updated to new incoming scenarios.

Software testing represents an important activity for SDCs, to make sure that the system being deployed has a certain degree of safety. Testing aims at revealing failures and adjusting the DL system such that in the next release the same failure will not occur [88]. For SDCs, software failures can have dramatic consequences and cause casualties, which further motivates the need for exhaustive testing.

1.1 Evolution of Self-Driving Cars

The first attempt to build an autonomous vehicle dates back to 1926, when the first remotely controlled (RC) car was invented. During the 1980s, computer vision and LIDAR were first used to build autonomous vehicles that could drive in a traffic-less road. In the late 1980s, semi-autonomous vehicles that could drive in the highway already achieved encouraging results, with thousands of kilometers run by the system at high speeds [70, 81].

In 2004, the first DARPA (Defense Advanced Research Projects Agency) Grand Challenge took place [5]. The goal of the original challenge was to build an autonomous vehicle capable of completing a 212 Km track of desert roadway. While no team managed to complete the tour that year, just the following year, in 2005, 5 cars managed to do so, achieving what was considered an incredible accomplishment. The DARPA Grand Challenges took place each year up until 2013, featuring since 2007 also urban scenarios [19].

In the broader area of computer vision, ImageNet is a dataset of over 15 million high-resolution images, featuring more than 22’000 image categories [22]. The *ImageNet Large-Scale Visual Recognition Challenge* (ILSVRC) [80] is an image classification challenge that uses a sub-set of the ImageNet dataset with roughly 1’000’000 images belonging to 1’000 different categories. The advent of deep learning dates back to 2012, when a deep learning model, a Deep Convolutional Neural Network (CNN) in particular, efficiently trained on two Graphical Processing Units (GPUs) managed to achieve state-of-the-art results on the ILSVRC [55]. Since then, an ever increasing amount of research has been conducted in the field of deep learning, with deep learning systems beating previous state-of-the-art technologies in a variety of fields, specially in computer vision tasks [23, 60]. CNNs in particular, had since been heavily used in SDCs to process the input images coming from the camera sensors.

To date, state-of-the-art SDCs such as Google’s Waymo and Tesla’s autopilots mostly use Deep Learning models, which make their cars capable of steering, accelerating and breaking in their lane, as well as automatically braking in case of frontal danger. Tesla’s autopilot is even capable of looking for a parking lot and park with no driver on board [68]. While such features are offered, a fully autonomous vehicle safer than a human driver cannot yet be built [1, 6].

1.1.1 Classification of Autonomous Driving Systems

To date, SDCs are classified by the level of their autonomy ranging from level 0 (least autonomy) to level 5 (maximum autonomy) according to the Society of Automotive Engineers J3016 standard [44]. A system is called an Automated Driving System (ADS) starting from level 3. Note that levels are incremental, meaning that self-driving features offered at a certain level, are also offered in all higher levels.

- **0 - No driving Automation:** The vehicle does not feature any self-driving functionality and the driver is fully responsible of the vehicle.
- **1 - Driver Assistance:** The system either controls the lateral or longitudinal motion control, whilst the driver performs the remainder.
- **2 - Partial Driving Automation:** The system controls both the lateral or longitudinal motion, but the driver still has to pay attention for objects and unexpected events and respond to those.
- **3 - Conditional Driving Automation:** The system also takes care for the detection of objects and unexpected events and responds appropriately. The driver is still required to be ready to take control over the system in case of fallback.
- **4 - High Driving Automation:** The driver is not expected to take control over the vehicle at any moment in time.
- **5 - Full Driving Automation:** The system is not limited by its operational design domain. The car can drive without a human driver on board. The system is also called 'driver-less'.

Note that in level 5, since no human presence is needed at all, the system is not only defined automated, but it is also defined autonomous.

1.1.2 Tasks of an Autonomous Driving System

The activity of driving a car is typically composed of four fundamental tasks [35]:

1. Localization
2. High-level path planning
3. Low-level path planning
4. Motion control

For each of these tasks, either more traditional rule-based algorithms or DL systems are used. If the whole pipeline is managed by DL systems, we call this system end-to-end (E2E). In such systems, the DL component learns all computations that need to be executed such that, given a raw input, the desired output is produced. In other words, the system learns the whole process at once, rather than learning just the individual steps and executing them sequentially.

Localization is the first task to be carried out. Starting from a source point and a destination point on a map, the SDC first figures out its relative position with respect to the surrounding environment. High-level path planning is the act of selecting a continuous path that the SDC must follow to reach the final destination. This technology is already consolidated in map services such as, for example, Google Maps. Low-level path planning concerns selecting the exact motions that the car will execute in order to continue its path towards the destination for any particular segment of the path, e.g. the strategy with which another vehicle is overtaken, how a difficult right-turn is executed, how and when the lane is changed, stopping at traffic lights, slowing down near pedestrians and so on. Finally, the motion control system is responsible for correcting errors that make the car deviate from the path trajectory previously decided.

In this work we consider the main task of lane keeping. Lane keeping is one of the essential components in SDCs, as it makes sure the vehicle is able to follow its assigned lane successfully when it is meant to, without invading other lanes and causing thus unwanted and dangerous situations. In particular, we study and compare online and offline testing of DL models performing the lane keeping task both in a simulated and a real environment. We analyze how their performances vary when trained and deployed in a simulator and in the real world. Note that our version of the task is a simplified version of what an actual SDC should deal with. In fact, through our studies, we do not deal with all the possible variables occurring on real driving scenarios such as pedestrians, traffic lights, weather conditions.

1.2 The Role of Simulators in Robotics and Autonomous Vehicles

In robotics, simulators are typically a cheap and convenient way both to collect virtual data on which DL models can be trained and tested [15, 104]. In the case of SDCs, collecting a "real" dataset consists in driving hundreds or even thousands of kilometers. This takes a long time and it is also expensive. Moreover, during testing, an insufficiently trained model may lead to expensive crashes or, even worse, to dangerous scenarios. Using simulators, however, both the data collection and testing processes can be sped up to save time and the only costs that are faced are energy-related. Furthermore, simulators provide a safe way to collect near-crash and recovery observations—that are essential to make the models more robust—and are rare to observe in real world scenarios.

The main drawback of simulators is that, in general, it is very hard to create a simulated environment that does reflect the real environment well enough from the perspective of a machine learning model. This causes models trained on virtual data to not being able to perform well in the real environment because simulated test sessions do not give enough insights on how the model may behave in the real world.

To cope with these problems, data-driven simulators manipulate data collected from the real world rather than creating a whole new physics-simulated environment [57]. This has the advantage of creating inputs that better resemble the inputs that the machine learning model will face when deployed. On the other hand, the number of such inputs is limited, whereas for a physics-based simulator there is in general no limit on the number of collectable inputs and they better reflect the states in which the vehicles will be in response to their actions.

Various simulators for SDCs are available. *NVIDIA Drive Sim* features photo-realistic images and has various industry partners [64]. *CARLA* [24] is an open-source simulator for research in the field of self-driving cars. *Udacity Simulator* [94] is yet another open-source simulator built on top of the *Unity* game engine [36]. *VISTA* (Virtual Image Synthesis and Transformation for Autonomy) instead is a data-driven simulator developed at MIT that was successfully used to learn policies that apply to real world roads [3].

Today, both Tesla and Waymo make use of a simulator to see how their models behave in critical situations [68].

1.3 Offline and Online Testing

Machine Learning models are typically tested by measuring their performances on an unseen test set of observations that was not previously used to train the model. Performance is measured using either some loss function, accuracy, recall, F1-score, false-positive rate, false-negatives rate or others. In this thesis, we will refer to this modality of testing as offline testing because the model is tested at a model-level, evaluating its predictions as a standalone component.

Online testing, on the other hand, refers to the practice of testing models as components within bigger software system and in the operational context in which they are supposed to be used. Thus, online testing evaluates the models based on the overall behavior or performance of the whole system. In the case of self-driving cars, online testing means evaluating the correctness of the predictions with respect to higher level requirements such as safety or user comfort [48].

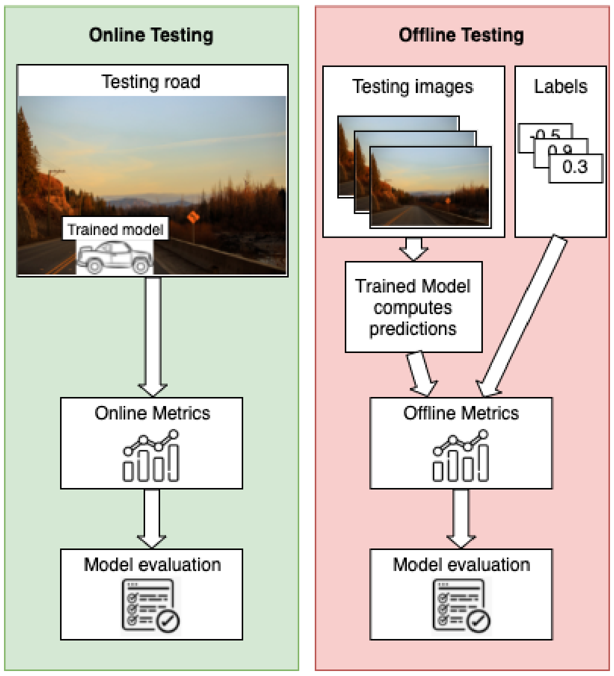


Figure 1.1. Schematic representation of online and offline testing

Figure 1.1 presents a schematic representation of online and offline testing methodologies. Offline testing (red box) measures the quality of a model based on the predictions made on any computer with no constraints on prediction times, while online testing (green box) measures the performance of a model based on online metrics collected while the model was driving the car and has received the inputs sequentially and rapidly.

Offline and Online Testing differ mainly in the fact that the former aims at finding the best model from a machine learning perspective, while the latter aims at finding the model that yields the best autonomous vehicle. Intuitively, if a model performs satisfactorily during offline testing setting, it is reasonable to believe that the model will also yield a good driving of the autonomous vehicle.

During online testing, it is also possible to observe how the chain of predictions has impact on the overall driving of the vehicle. Imperceptible errors done at the prediction level may add up to a substantial amount, causing system-level failures that are not observable during offline testing. Since the stream of inputs fed to the model is sequential, in online testing we can see whether the model is robust to minimal deviations to the trajectory of the car or whether the vehicle deviates more and more from the its lane as soon as some apparently minor error in the prediction is made. Thus, during online testing, we can observe whether the system can compensate for errors, while in offline testing this information cannot be observed.

However, online testing for real world cars is both expensive and potentially dangerous. A simulator provides a cheap and safe way of testing a model online, at the cost of not knowing whether the model trained for the simulated environment will actually adapt to the real world. In the literature, substantial work has been carried out to study online and offline testing of self-driving cars in simulated environments [48, 84, 37, 30, 75]. However, due to its costs and risks, little research has been done for online testing in the real world.

1.4 Research Questions

With this thesis, our aim is to answer the following research questions:

- **RQ1:** *How do offline testing metrics in a simulated environment compare to the same metrics in a real environment?*

- **RQ2:** *How do online testing metrics in a simulated environment compare to the same metrics in a real environment?*

- **RQ3:** *Can offline testing replace online testing?*

1.4.1 Research Question One

How do offline testing metrics in a simulated environment compare to the same metrics in a real environment?

While offline testing is generally possible for real world models, simulation also provides offline testing metrics. One possible intuition that can be used to assess quality of real world models is the following: if a model that drives well in simulation has similar offline testing performances for both environments, then also the real world model shall perform well in the real world.

Given two comparable simulated and real environments, does offline testing perform similarly in both environments? Or do some differences arise? If so, how important are these difference?

We answer this research question by comparing offline testing metrics of the same model architectures both in our simulated and real environment.

1.4.2 Research Question Two

How do online testing metrics in a simulated environment compare to the same metrics in a real environment?

In the field of SDCs, physics-based simulators are often used for online testing as they provide cheap and safe testing. The intuition is that if a model trained on simulated data performs well in the simulated environment, then the same architecture trained on a similar amount of real world data should yield similar performances in the real world. However, it is not clear to what extent this intuition holds.

To answer our second research question, we compare the driving performance of the same model architectures in our real and simulated environments.

1.4.3 Research Question Three

Can offline testing replace online testing?

Offline testing typically acts as a "proxy" for online testing, meaning that models that perform poorly in isolation are not expected to yield a well driving SDC. However, models that perform relatively well in offline testing can have very different driving behaviors when tested online.

Our last research question asks whether we could tell apriori, by just testing models offline and without testing them in the online setting, which self-driving model yields the best driving SDC. Another way to phrase this question is "If we rank DNN models by their offline testing performances, will the ranking order hold when we consider online testing performance?".

We answer this research question comparing the suggestions coming from offline testing with the actual driving performances revealed with online testing. We do so for both a real and a simulated environment.

If the answer to this question is "Yes", then online testing, which is more expensive and potentially dangerous with respect to offline testing, could be avoided in most cases.

1.5 Thesis Structure

The organization of the document is as follows.

- Chapter 1 introduces the thesis work.
- Chapter 2 presents the related work. We refer to works that are related in terms of testing of autonomous vehicles, in the use of a simulator, and in the use of the DONKEY CAR.
- Chapter 3 provides the reader with the required background, which covers supervised learning, Convolutional Neural Networks (CNN) and Generative Adversarial Networks (GAN).
- Chapter 4 describes our first contribution by overviewing currently available open-source projects of small-scale self-driving cars and comparing them under different aspects, such as the availability of simulator, costs and more.
- Chapter 5 presents the DONKEY CAR framework and all functionalities implemented by the time of our studies by the open-source community, including code to collect data, train and deploy models both in the real world and in the simulator. We also present our own version of the donkey car.
- Chapter 6 and Chapter 7 describe the simulated and real environment respectively, i.e., our experimental setup. We also describe the improvements we made on the DONKEY CAR simulator.
- Chapter 8 presents our experiments and obtained results for different machine learning models both in the simulator and in the real world.
- Chapter 9 illustrates our conclusions and future work.

Chapter 2

Related Work

In this chapter, we discuss the works related to this thesis. In particular, we provide the reader with an overview of studies about online and offline testing for machine learning models, testing of SDCs, testing of SDCs through GANs, attempts to use simulated data to train real world SDCs and finally some studies conducted with the DONKEY CAR platform. For larger treatments of the software testing practices and concepts for machine learning, we refer the reader to the relevant surveys [75, 82, 9].

2.1 Offline Testing and Online Testing of SDCs

Codevilla et al. [17] find that machine learning models with good offline testing performances are not correlated with driving quality. Also, they find that models with similar offline testing performances can differ dramatically in driving style when deployed on the vehicle.

In their work, Haq et al. [37] study whether simulated data can be used as substitute for real world data and how online and offline testing differ from and complement each other using self-driving cars as study case. They find offline testing to be more optimistic with respect to online testing, where inaccurate steering predictions accumulate and can have a butterfly effect. Moreover, they argue that simulated data can yield prediction errors similarly as real world data can, and that thus simulated data can be used for testing.

In the extension of the previous work, Haq et al. [38] confirm, using simulation, that online and offline testing differ in that offline testing is more optimistic. The authors find weak correlation between offline prediction and online performance. The authors also argue that simulated data can be used as replacement of real world data for testing of DNNs, as the difference in prediction errors is on average relatively low.

Similarly to these works, in our work we found that good offline testing performance does not necessarily translates to good driving quality. Differently, we also consider the offline/online testing of SDCs using real world, instead of simulated, data.

2.2 Testing of DNN-based SDCs

A variety of different testing techniques for DNN-based SDCs have been proposed. White-box techniques use information “within” the DL model (e.g., layer activations), while black-box testing techniques treat the DL component as a “black-box” and are architecture independent.

Jahangirova et al. [48] analyse hundreds of metrics used to assess the quality of driving of an SDC in a simulator, which were then compared against the human perception of "good driving". The study shows that the metrics that were most correlated with the human perception of quality driving can be used as functional oracles. Some of the metrics, such as the standard deviation of the steering angle and the standard deviation of speed, are also used in this thesis. Stocco et al. [85] propose SelfOracle, an approach based on auto-encoders to estimate the black-box confidence of the DNN-based SDC at runtime. Results show that SelfOracle is able to predict 77% of cases of misbehavior up to 6 seconds in advance.

Zhou et al. [102] present DeepBillboard, a tool to generate adversarial examples featuring drive-by billboards for real self-driving. Adversarial examples are generated both physically and digitally. DeepBillboard is capable of misleading SDCs by increasing their average steering angle error by up to 26.44 degrees. Tian et al. [91] present DeepTest, a white-box testing system capable of automatically detecting erroneous behaviors of known steering angle regression architectures from the Udacity challenge such as Epoch, Rambo and Chauffeur [12, 11].

Gambi et al. [30] use procedural content generation to automatically create test cases for SDCs in a simulator. Their AsFault tool produces more challenging scenarios than those obtained with random testing. Riccio and Tonella [74] present DeepJanus, a tool capable of searching for frontier inputs for DL systems. The authors show that well-trained DL systems have larger frontiers than poorly trained models.

While the above mentioned works focus on producing difficult test cases for DL systems or predicting when and whether the system will misbehave, in our work we compare how online and offline testing compare with each other, without altering the original test scenarios.

2.3 Autonomous Vehicles Testing with GANs

Lee et al. [56] try to determine if an input image fed to a machine learning model is out-of-distribution and thus less confidence should be assumed by the model. They do so by simultaneously optimizing a generative model (both generator and discriminator) and the machine learning classifier using the confidence loss defined by the authors. Similarly, Nitsch et al. [63] also find out-of-distribution inputs by studying the case of self-driving cars and applying some minor improvements to the work by Lee et al.

Xia et al. [98] with their *SynthCP* use a segmentation model M to obtain a segmented version of the SDC’s input image. They then feed the segmented image to a Conditional GAN trained to convert such images into photo-realiastic road images. They then use the discrepancy between the original image and the reconstructed image to detect failures and anomalies.

Finally, Zhang et al. [100] use the *DeepRoad* unsupervised framework to generate artificial road images with different meteorological conditions that can be used to further train SDCs models. Similarly to our work, the authors are able to translate images from a distribution to another and vice-versa by using the UNIT framework [58]. They use the UNIT framework to translate images featuring sunny weather conditions to rainy weather conditions and vice-versa.

UNIT learns two variational autoencoders, two generator networks and two discriminator networks. The two variational autoencoders are used to map input images from the two distributions to a common latent space. The common latent space embeds semantic content. The two generators can generate images starting from a vector in the latent space. Each generator generates differently styled images, but with similar semantic content. Similarly to CycleGAN [103], UNIT learns by using a cycle-consistency loss term in the total loss function.

The work by Zhang et al. [100] is very related to our work in the use of GAN for testing. Hereafter, we highlight the main differences. First, we translate images using CycleGAN rather than UNIT. While with UNIT it is possible to find a common latent space which can be manipulated, CycleGAN is more capable of producting realistic results and requires no paired dataset, as argued by Liu et al. [97]. Furthermore, we translate simulated data into real data and vice-versa, while Zhang et al. [100] use real world data only which are expensive to collect. Translating simulated data into real world images is, to the best of our knowledge, a novel approach. Indeed, while Zhang et al. [100] use UNIT for test generation, we use a GAN to train quality metrics that are difficult to obtain for real data. The benefit of such approach is the possibility to cheaply create realistic images paired with quite accurate quality indicators.

2.4 From simulated to real environments

Amini et al. [3] use their data-driven VISTA simulator to train a RL agent which is then deployed on a real self-driving car moving at a constant speed. Their simulator generates photo-realistic images starting from a real world dataset and applying perspective changes to the images such as shifts and rotations by first projecting the images in a 3-dimensional space, applying the necessary transformations and then projecting back to a 2-dimensional space. The authors also train an imitation learning model on the CARLA simulator and try to fill the gap with the real world sensed images with domain randomization and domain adaptation (in the simulator). After comparison, the authors find their RL model to be more capable of recovering from critical situations w.r.t the state-of-the-art supervised learning algorithms.

Similarly, NVIDIA's PilotNet by Bojarski et al. [8] models are tested using a closed-loop testing based on real sensor recordings rather than synthetic data. Also, they use an in-car monitor that displays PilotNet's performances which in turn helps humans with the development. Bojarski et al. [7] also successfully trained a CNN model for the task of self-driving by also using a data-driven simulator to test their model before deploying it on a real vehicle.

Most recently, Fremont et al. [28] use formal methods such as the *Scenic* scenario specification language and *VerifAI* [25] to generate testing scenarios that are executed both in simulated and real test environments. The simulated environments are produced with the open-source *LGSVL* simulator [77]. The authors conclude that simulation can provide insights on which real world test cases should be tested since they could reveal unsafe behaviours of the SDC. However, the authors also argue that mismatches between simulated and real world environments can lead to discrepancies. To measure such discrepancies, the authors use the Skorokhod and the Dynamic Time Warping (DTW) metrics.

Tremblay et al. [93] use domain randomization on the simulated data (altering simulator lightning, pose, textures and more) to create a more robust classifier for the task of object detection. They find that a model trained both with simulated and real world data outperforms the same model trained on real world data only.

2.5 Studies with DonkeyCar

Studying self-driving cars through remotely controlled small scale cars is a safe and inexpensive way to empirically test hypothesis that might hold for real cars [83].

Previous works use DONKEY CAR to train reinforcement-learning (RL) systems and analyze how well such systems behave. Zhang et al. [101] use the Double-Deep Q-Network algorithm (DDQN) to train an agent on the donkey simulator and then test it on the corresponding real world track. They do so by heavily preprocessing the input camera frame, segmenting the images as to only display the lanes with computer vision tools such as Canny Edge Detection and Hough Transform. Viitala et al. [96] implement a reward mechanism in the real world and then compare how different RL models learn both in the donkey simulator and in the real world (but never transferring a simulated model into the real world). The authors make use of the soft actor-critic (SAC) algorithm and a Variational Autoencoder (VAE) for state representation to train 3 differently initialized models (fixed VAE, pre-trained VAE, and non-initialized VAE). In the simulator, they use randomly generated tracks, while for the real world, they use a single custom track. They find that all of their models learn to drive in approximatively 10 minutes, and that training a VAE from scratch works best in the real world. Also, they find that the simulator dynamics are much simplified with respect to the real world, as the throttle of the real car is hard to control due to dependency on battery voltage, temperature and other factors.

Similarly to Viitala et al. [96] we find that the dynamics of the simulator do not fully match the real world. Unlike the above listed works, however, in this thesis we focus on the supervised learning setting, which is the mainly used approach to self-driving nowadays. Furthermore, unlike the previous works, we benefit from a setting where simulated and real world environments are developed jointly. This allows us to study differences between simulation and the real world keeping the environmental variable comparable.

Chapter 3

Background

Today, the most successful approaches in building an SDC are based on Deep Learning (DL). Both the supervised learning and the reinforcement learning settings have been explored for building such vehicles.

In this chapter, we provide the reader with an overview of the concepts and algorithms used in the document. Particularly: Supervised Learning, Convolutional Neural Networks (CNNs), Generative Adversarial Networks (GANs), Conditional Generative Adversarial Networks (cGANs) and the known CNN architectures for self-driving that we are going to use for our experiments in Chapter 8.

3.1 Supervised Learning

In supervised learning, the goal is to learn a function f_θ that maps a set of training points (observations) to the relative targets (labels), where θ is the set of parameters to be optimized [39, 21]. In particular, function f_θ should be able to generalize to unseen inputs. To do so, the function is tested against a set of observation-label pairs that has not been used for learning f_θ. The set of observations and labels used to train the model is called the training set, while the set of observations and labels used to test the obtained model is called test set. A third set, called the validation set, can be used to stop the training algorithm when the model starts under-performing on such set. Labels can be either continuous values, in which case the task is called *regression*, or discrete values, in which case it is called *classification*.

In the case of SDCs, multiple learned functions f_θ are used at driving time for a multitude of sub-tasks and the associated inferences are used to control the car. For example, Tesla uses 48 such models where each of the models is optimized for a particular task like stop sign detection, traffic lights recognition or lane keeping [50].

In our work we focus on the regression tasks, where the machine learning model has to learn continuous values for steering and throttle given the input image obtained through a front-facing camera.

DL models learn function f_θ by minimizing what is called a loss function [71, 99]. A loss function, typically denoted as $\mathscr{L}$, captures the discrepancy between the model predictions and the desired output. Adapting the model parameters θ as to minimize the loss function makes the model more capable of producing the right outputs for the training data and, hopefully, for some unseen data. The most common loss function for regression tasks is the Mean-Squared Error (MSE) defined as:

$$\mathscr{L}(\theta) = \frac{1}{N}\sum_{i}^{N}(y_i - f_\theta(x_i))^2$$

Where N is the number of training examples, and (x_i, y_i) are the i-th observation and the i-th label respectively. In the case of self-driving cars, x_i would be the set of inputs perceived by the car at a given moment, and y_i corresponds to the desired output given those inputs, also called ground-truth. Typically, such ground-truth is collected manually by human drivers.

Although various algorithms exist to find f_θ, nowadays state-of-the-art results are obtained using mainly Deep Neural Networks (DNNs), Convolutional Neural Networks in particular, optimized through stochastic gradient descent (SGD) and back-propagation (BP). Also note that DNNs and BP are used outside of the supervised learning context too.

3.1.1 Deep Neural Networks

The most basic kind of Artificial Deep Neural Network is referred to as Feed-Forward Neural Network (FFNN) or Multi-layer Perceptron (MLP). FFNNs are general function approximators and are inspired by the human brain, in that they features artificial neurons connected through edges.

Deep Neural Networks transform an input vector $x_i \in \mathbb{R}^d$ to an output vector $\hat{y}_i$ by applying a multitude of computations performed within each layer. The "deep" term in DNN refers to the depth in the number of layers of the network.

Assuming the network is composed of L layers, the output of the network is obtained as:

$$\hat{y} = a_L \circ l_L \circ a_{L-1} \circ l_{L-1} \circ \ldots \circ a_1 \circ l_1(x)$$

where l_i is the i-th linear function, typically represented as a matrix-vector product, and a_i is the i-th non-linear activation function, typically an element-wise operation on the resulting vector. The linear transformation happening in layer i can be described with a matrix of weights $W_i \in \mathbb{R}^{d_l x d_{l-1}}$, where d_{l-1} is the dimension of the vector after going through the previous layer (with $d_0 = d$ the dimension of vector x), and d_l is the dimension of the internal representation after going through layer l. We thus have that:

$$l_i(x) = W_i x$$

Popular activation functions include ReLU, sigmoid, tanh and others [20]. The ReLU activation function has been proven to make training converge much faster with respect to other activation functions as the gradients flow through the network by vanishing less [65].

Such non-linearities are essential to obtain models which are capable of expressive power and thus learning more complicated patterns. In fact, if such activation functions were not present, the model would be reducible to a linear function, since a composition of linear functions remains a linear function.

For efficiency reasons, multiple output vectors $\hat{y}_i$ are computed by stacking vectors x_i in a single matrix X, where each row represent a training point. The outputs are then obtained as:

$$\hat{Y} = a_L \circ l_L \circ a_{L-1} \circ l_{L-1} \circ \ldots \circ a_1 \circ l_1(X^T)$$

Where $\hat{Y}$ is a matrix containing all output vectors in the columns. Note that in feed-forward neural network, the set of parameters to be learned is the set of weights $W_i \forall i \in [1, ..., L]$.

3.1.2 Gradient Descent and Stochastic Gradient Descent

Gradient Descent is an optimization technique that can be applied on differentiable loss functions [52]. The idea is to iteratively find a better set of parameters θ by computing the gradient of the loss function with respect to θ and to update the parameters in the opposite direction of the gradient as follows:

$$\theta' = \theta - \eta \frac{\partial}{\partial \theta} \mathscr{L}(\theta)$$

Where θ are the current parameters, θ' represents the new set of parameters and η is a small constant which represent the step size in the opposite direction of the gradient. With the updated set of parameters, if the step size η is small enough, and if parameters θ did not represent a local minima already, we will have that$\mathscr{L}(\theta') < \mathscr{L}(\theta)$. Updating parameters iteratively in this manner, the algorithm converges to a locally optimal set of parameters.

Although effective, Gradient Descent makes the training procedure slower as the size of the training set increases due to the fact that the loss function has to be computed for every training observation-label pair.

For this reason, in practice it's is preferable to compute approximations of the real gradient $\frac{\partial}{\partial \theta}\mathscr{L}(\theta)$ by only considering a randomly selected sub-set of training points and iterating them in small batches. This technique is called Stochastic Gradient Descent (SGD) since, in fact, gradients are stochastic. SGD has been proven to converge much more quickly with respect to Gradient Descent to a set of locally optimal parameters θ.

Finally, different optimization algorithms have been developed to avoid getting stuck in settle-points when the gradient approaches zero and to converge more quickly to a good set of parameters [86, 29].

3.1.3 Back-Propagation

Back-propagation is an efficient algorithm to compute gradients in neural networks [41, 76]. Such gradients are typically then used to update the set of parameters θ to be optimized, but can in principle be used for other purposes, such as generating adversarial examples [34], generating images with particular features [31] and more.

BP makes use of the chain-rule of differentiation, which states:

$$[f \circ g]' = [f' \circ g]g'$$

or more generally, for a composition of n functions the derivative is:

$$[f_1 \circ f_2 ... \circ f_n]' = \prod_i^n [f_i' \circ f_{i+1} ... \circ f_n]$$

Back-propagation applies the chain-rule starting from the last layer l_L and computing all gradients until even the gradient of l_1 is computed. The efficiency of the algorithm lays in the fact that partial derivatives computed in deeper layers are re-used to compute partial derivatives of previous layers.

For example, in the case of a feed-forward neural network, the gradient of the loss function $\mathscr{L}$ with respect to the matrix W_L is computed as:

$$\frac{\partial \mathscr{L}}{\partial W_L} = \frac{\partial \mathscr{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial l_L} \frac{\partial l_L}{\partial W_L}$$

The back-propagation algorithm then computes the gradient of the loss function with respect to matrix W_{L-1} by exploiting the already useful partial derivatives $\frac{\partial \mathscr{L}}{\partial \hat{y}}$ and $\frac{\partial \hat{y}}{\partial l_L}$:

The re-use of partial derivatives makes the algorithm especially efficient for deep networks with many layers.

$$\frac{\partial \mathscr{L}}{\partial W_{L-1}} = \frac{\partial \mathscr{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial l_L} \frac{\partial l_L}{\partial a_{L-1}} \frac{\partial a_{L-1}}{\partial l_{L-1}} \frac{\partial l_{L-1}}{\partial W_{L-1}}$$

3.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a special kind of neural networks in which convolution operations are performed within each layer [69, 2].

CNNs are heavily used for tasks of image-classification, image segmentation, object detection, computer-vision and more [16, 10, 4].

3.2.1 Two-dimensional Convolution

In image processing, convolution refers to the act of filtering a source image through the use of a filter / kernel. This operation can reveal patterns in the source image, such as borders and edges.

Mathematically speaking, if the source image I^{src} has width W^{src}, height H^{src} and depth (typically in color channels) D, the result of applying a convolution using kernel K which has width W^{ker}, height H^{ker} and depth D is an output image I^{out}. The kernel is "slid over" I^{src} such that the operation

$$\sum_{a}^{W^{ker}} \sum_{b}^{H^{ker}} \sum_{c}^{D} I^{src}_{i+a,j+b,c} K_{a,b,c}$$

is carried out for all source coordinates (i, j) such that $i + W^{ker} \leq W^{src}$ and $j + H^{ker} \leq H^{src}$.

The resulting image I^{out} of such operation is two dimensional. To have a three dimensional output image, it is sufficient to apply multiple kernels K and "stack" the results together in the new dimension. We refer to these matrices stacked together, and to all multi-dimensional parallelepipedal-shaped collection of values, as a tensor.

In a two-dimensional convolutional layer of a CNN, the source image is mapped to a (typically) deeper matrix and multiple kernels are used by the neural network. The tensor is also called a feature map.

In CNNs, multiple convolutional layers are used in sequence. Furthermore, after each convolution a non-linearity is applied to the just obtained feature map.

Notice that in CNN, kernels are the parameters to be learned. The final feature map is then typically "flattened" into a vector of values, which is then fed to a FFNN for classification or regression.

Practitioners typically just define the number and dimension of the kernels in each convolutional layer, while the neural network learns the value of each component of the kernels.

Finally, altering padding and stride allows to shape the output tensor arbitrarily [26]. Padding consists in enlarging the input tensor in width and height with arbitrary values. The stride is an integer value that specifies by how much should kernels be slid over the input tensor. Notice that for both padding and stride, different values can be used horizontally and vertically.

3.2.2 Pooling

The pooling operation is used in CNN to both reduce the number of parameters and complexity of the model, and to make the model more robust to variations in the position of the features [32]. Given an input feature map, the pooling operation shrinks the feature map along the width and height dimension by applying the same kernel as many times as the depth of the input tensor. Usually, such kernel uses a stride bigger than one. Examples of such kernels are max-pooling, where the kernel performs a max operation, or average-pooling, where the kernel performs an averaging operation.

3.3 Architectures for Self-Driving Cars

As for other tasks in machine learning, researchers have published the architecture and implementation of their models trained to drive a self-driving car. In the following, we describe the architectures of the fours CNN models that will be used for our empirical experiments, namely DAVE-2 [7], the default DONKEY CAR model, Chauffeur [12] and Epoch [89].

DAVE-2 is a popular network architecture proposed by Bojarski et al. back in 2016 [7]. The network was shown capable of autonomous driving 98% of the time, according to the authors. All of the other models we study in this thesis have drawn inspiration from the DAVE-2 model. The DONKEY CAR default model is a CNN much similar to the DAVE-2 Model, except for a higher number of convolutional filters. Chauffeur is one of Udacity's community open source models for steering angle prediction. The model classified as third best model in the *Udacity Challenge 2* in 2016 [11, 12]. Similarly, the Epoch model ranked 6th in the same challenge. Both models, unlike DAVE-2 and DONKEY CAR's default model, feature pooling layers.

We slightly modify the model architectures as to have two output neurons in the final layer, since in our setting we are making regression on both the steering and throttle commands. Also, for all models, we fixed the size of the input image to 140x320 (3 color channels). We will clarify the reason for this design choice in Chapter 8. Finally, we further adapted the Epoch and Chauffeur architectures to better fit our experimental setup and to match the memory constraints imposed by the small computer memory of the physical car.

3.3.1 DAVE-2

The DAVE-2 model by Bojarski et al. [7] is among the most known open-source architectures in the self-driving car literature.

The DAVE-2 network first normalizes the input image by subtracting the mean of pixels and dividing by the standard deviation. The normalized image is then passed through three convolutional layers of size 5x5 (24, 36 and 48 kernels respectively) and then through two convolutional layers of size 3x3 (64 kernels each). The result is then flattened and passed through

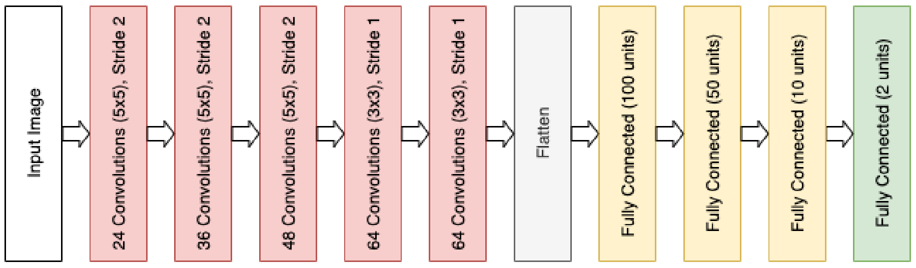

Input Image
24 Convolutions (5x5), Stride 2
36 Convolutions (5x5), Stride 2
48 Convolutions (5x5), Stride 2
64 Convolutions (3x3), Stride 1
64 Convolutions (3x3), Stride 1
Flatten
Fully Connected (100 units)
Fully Connected (50 units)
Fully Connected (10 units)
Fully Connected (2 units)

Figure 3.1. DAVE-2 architecture

four fully connected layers (100, 50, 10 and 2 units respectively). Each layer is activated with the Elu activation function. Figure 3.1 shows the architecture of our version of the DAVE-2 model. The input image is a normalized version of the image collected by the camera mounted on the car. Three 5x5 convolutions with stride 2 are followed by two 3x3 convolutions with stride 1 and a fully connected part. The activation function used after each convolution and linear layer is Elu.

The network was designed with an input of size 66x200 with 3 color channels in mind and only one output neuron. However, considering that we will be dealing with inputs of size 140x320 pixels and the fact that the final layer features one extra neuron with respect to the original architecture, our adaptation of DAVE-2 has a total of 2’249’030 trainable parameters.

3.3.2 DonkeyCar’s Default Model

The DONKEY CAR project allows to use different types of architectures. In our studies we focus on the default architecture of the project, which is a CNN very similar to the DAVE-2 architecture with some minor changes.

The default model architecture, similarly to the DAVE-2 architeture, passes the input image through three convolutional layers of size 5x5 (24, 32 and 64 kernels) and then through two convolutional layers of size 3x3 (64 kernels each). The result is then flattened and passed through three fully connected layers (100, 50 and 2 units respectively). Each layer is activated with the ReLU activation function.

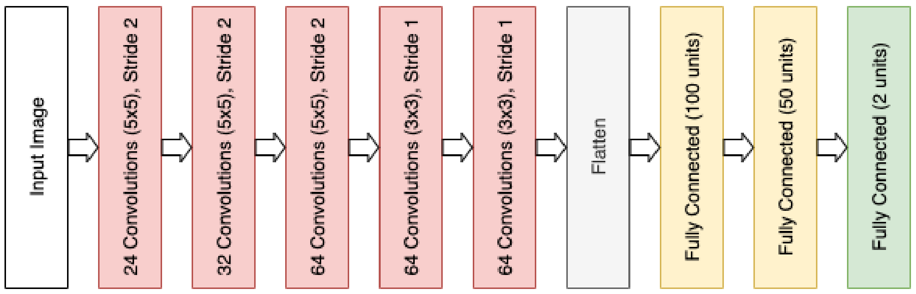

Input Image
24 Convolutions (5x5), Stride 2
32 Convolutions (5x5), Stride 2
64 Convolutions (5x5), Stride 2
64 Convolutions (3x3), Stride 1
64 Convolutions (3x3), Stride 1
Flatten
Fully Connected (100 units)
Fully Connected (50 units)
Fully Connected (2 units)

Figure 3.2. DonkeyCar default model architecture

Figure 3.2 depicts the default DONKEY CAR model. Three 5x5 convolutions with stride 2 are followed by two 3x3 convolutions with stride 1 and a fully connected part. The activation function used after each convolution and linear layer is ReLU. During training, a dropout of 10% is applied after each layer.

By Figure 3.2 we notice that the DONKEY CAR's default model architecture differs only marginally from the DAVE-2 architecture. More filters are learned in the third convolutional layer, while slightly less connections are present in the fully connected part of the network. Importantly, also the activation function differs. The default model uses a ReLU activation function after each layer, rather than an Elu activation function.

3.3.3 Chauffeur

The Chauffeur architecture was inspired by NVIDIA's DAVE-2 model. Many different versions of this model are implemented in the Udacity community models. Although some versions of Chauffeur use Long Short-Term Memory (LSTM) cells [43] and encoders, we decided to use the CNN version.

The input image is processed by five convolutional layers, each followed by a 2x2 max-pooling operation. The convolutional layers have 16, 20, 40, 60 and 80 learnable kernels of size 5x5, 5x5, 3x3, 3x3, and 2x2 respectively. The resulting feature map is flattened and a linear operation is carried out to obtain the predictions for steering and throttle.

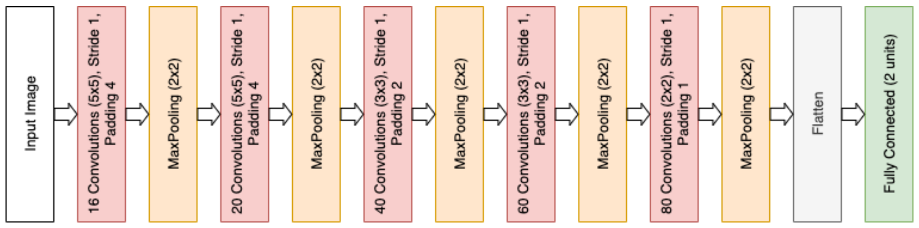

Input Image
16 Convolutions (5x5), Stride 1, Padding 4
MaxPooling (2x2)
20 Convolutions (5x5), Stride 1, Padding 4
MaxPooling (2x2)
40 Convolutions (3x3), Stride 1, Padding 2
MaxPooling (2x2)
60 Convolutions (3x3), Stride 1, Padding 2
MaxPooling (2x2)
80 Convolutions (2x2), Stride 1, Padding 1
MaxPooling (2x2)
Flatten
Fully Connected (2 units)

Figure 3.3. Chauffeur architecture

In Figure 3.3 we present our version of the Chauffeur model. Five convolutional layers , activated with the ReLU function, are each followed by a max-pooling operator of size 2x2. The feature maps are then flattened into a vector and a linear mapping is made to obtain predictions of steering and throttle. After each convolutional layer, a spatial dropout of 10% is carried out [92]. This is omitted in the image.

Chauffeur, with his 63'818 variables, is the model which features the least amount of learnable parameters.

3.3.4 Epoch

We adapt Epoch, from Udacity's community models, to have less learnable kernels in its convolutions and less neurons in the fully connected part of the network. In particular, we halve the number of learnable kernels and neurons in the second-to-last fully connected layer.

We decide to reduce the number of learnable parameters as we have found that the original Epoch model could not be entirely loaded into our car computer main memory.

Our version of epoch features thus three 3x3 convolutional layers with 16, 32 and 64 kernels respectively, each followed by a 2x2 max-pooling layer. The result is then flattened and connected to two fully connected layers of 512 and 2 units. The activation function used after each convolution is ReLU. During training, we use drop-out as regularization. We drop 25% of

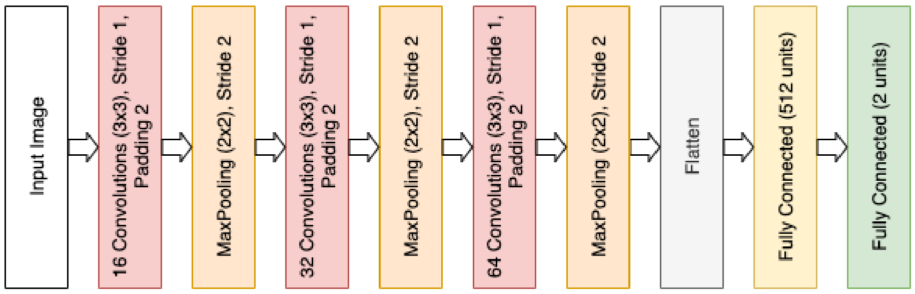

Input Image
16 Convolutions (3x3), Stride 1, Padding 2
MaxPooling (2x2), Stride 2
32 Convolutions (3x3), Stride 1, Padding 2
MaxPooling (2x2), Stride 2
64 Convolutions (3x3), Stride 1, Padding 2
MaxPooling (2x2), Stride 2
Flatten
Fully Connected (512 units)
Fully Connected (2 units)

Figure 3.4. Epoch architecture

the data after the first two max-pooling operations and 50% after the third max-pooling and the 512-units fully-connected layer.

Figure 3.4 is an illustration of our modified version of the Epoch model. 3x3 convolutions are followed by Max-Pooling layers with stride 1 and padding 2. The result is then flattened and fed into a two layers fully-connected network. During training, dropout of 25% is used after the first two max-pool operators and of 50% after the third max-pool operator and first fully-connected layer. Each layer is activated by the ReLU activation function except for the max-pool operations and the final layer.

Epoch is the model with the highest number of learnable parameters in our studies, due to the high number of units in the first fully connected layer. The model has in fact 22'307'362 learnable parameters.

3.4 Generative Adversarial Networks

Generative Adversarial Networks (GANs), first proposed by Goodfellow et al. in 2014 [33], are generative models capable of producing adversarial and realistic inputs.

GANs are composed of a Generator network G and discriminator network D that compete on the same loss function, that is, G tries to minimize the loss function while D tries to maximize it. The role of G is to generate realistic observations, typically images, while the role of the discriminator D is to discriminate between real images and images that were generated by G.

As the training procedure goes on, network D becomes more capable of distinguishing real from artificial data and, in turn, network G learns to generate more realistic samples.

Optimal generator and discriminator are found by finding the generator that minimizes and the discriminator that maximizes the value function V(D, G) :

$$\min_G \max_D V(D,G) = \mathop{\mathbb{E}}_{x\sim p_{data}(x)}[log(D(x))] + \mathop{\mathbb{E}}_{z\sim p_z(z)}[log(1-D(G(z)))]$$

where p_{data} is the probability distribution of real data, and p_z is a prior on noise variables z , which lie in the latent space of G.

3.4.1 Conditional Generative Adversarial Networks

Conditional GANs are an extension of GANs. Unlike regular GANs, the generator and discriminator of Conditional GANs not only take as input a random noise vector z or the generated image $G(z)$, but also some additional information which directs the data-generation process [59]. Such information is, in our studies, an image to be transformed or mutated.

Conditional GANs are popularly used for tasks of image-to-image translation like colorization, background masking, style-mixing, and more [62, 95, 14]. For image-to-image translation tasks, the additional information put in the generator model is a source image which the model can modify to obtain an image that has the same semantic content, but looks as drawn from another distribution.

The “Pix2Pix” model by Isola et al. [46] is a popular conditional GAN whose architecture is based on U-Net [78] and uses batch-normalization layers [45]. The network is fed with an input image x and a noise vector z, and in turn generates an output image which semantically resembles x but looks drawn from a target distribution.

3.4.2 CycleGAN

CycleGAN, by Zhu et al. [103], is a conditional GAN that allows to learn two image-to-image translators starting from an unpaired dataset. The work builds on the framework of Pix2Pix [46].

The main advantage of using CycleGAN over other image-to-image generative models is that CycleGAN can be trained even in the absence of a paired-dataset, which is often hard and expensive to collect. It is in fact possible to learn, starting from data drawn from two different distributions, a function that converts an observation from the first distribution to an artificial observation that looks as if it was drawn from the second distribution and, respectively, a function that does the opposite operation. The two learned functions are also optimized such that if an input is run through both functions, the result looks similar to the original input itself.

More precisely, let us assume having images distributed according to two distributions X and Y. The CycleGAN architecture allows to find functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$ such that for inputs $x \sim X$ and $y \sim Y$:

- $G(x) = \hat{y}$ is indistinguishable from other samples in Y according to discriminator D_Y
- $F(y) = \hat{x}$ is indistinguishable from other samples in X according to discriminator D_X
- $F(G(x)) \approx x$, i.e. function F is roughly inverse of G (forward cycle-consistency)
- $G(F(y)) \approx y$, i.e. function G is roughly inverse of F (backward cycle-consistency)

The last two conditions combined form the so called *cycle consistency*, which follows the intuition that functions F and G should be each other's opposite and which discourages the generating functions F and G from always generating the same outputs.

The loss function that allows to find functions F and G such that all previously mentioned conditions hold is a combination of two normal adversarial losses and cycle consistency loss:

$$\mathcal{L}(G, F, D_X, D_Y) = \mathcal{L}_{GAN}(G, D_Y, X, Y) + \mathcal{L}_{GAN}(F, D_X, Y, X) + \lambda \mathcal{L}_{cyc}(G, F)$$

where:

- $\mathcal{L}_{GAN}(G, D_Y, X, Y) = \mathbb{E}_{y \sim Y}[\log(D_Y(y))] + \mathbb{E}_{x \sim X}[\log(1 - D_Y(G(x)))]$
- $\mathcal{L}_{GAN}(F, D_X, X, Y) = \mathbb{E}_{x \sim X}[\log(D_X(x))] + \mathbb{E}_{y \sim Y}[\log(1 - D_X(F(y)))]$
- $\mathcal{L}_{cyc}(G, F) = \mathbb{E}_{x \sim X}[||F(G(x)) - x||_1] + \mathbb{E}_{y \sim Y}[||G(F(y)) - y||_1]$

- λ is a scalar hyper-parameter that enforces the cycle consistency loss

Note that is $\mathscr{L}_{GAN}(G, D_Y, X, Y)$ and $\mathscr{L}_{GAN}(F, D_X, X, Y)$ are normal adversarial losses that allow to find functions G and F respectively, while $\mathscr{L}_{cyc}(G, F)$ is the cycle consistency loss that motivates functions F and G to produce various different outputs and to be each-other's approximate inverse. Optimal functions F^* and G^* are then found as

$$G^*, F^* = \arg\min_{G,F}\max_{D_X,D_Y} \mathscr{L}(G, F, D_X, D_Y)$$

typically through stochastic gradient descent and back-propagation. The two generators G and F share the same architecture, which is proposed by Johnson et al. [49].

Chapter 4

Self-driving RC car open-source projects

For the purpose of this study, we decided to reproduce in the small the behaviour of a self-driving vehicle by using a physical remote control (RC) car equipped with at least one camera sensor and a small-sized computer. Indeed, the DNN-based autopilots described in Chapter 3.3 process imagery inputs from a front-facing camera and output the vehicle's actuators values such as speed and throttle. Ideally, the chosen platform would also provide an open-source simulator capable of reproducing the functioning of the RC vehicle with the highest possible fidelity.

With these two requirements in mind, we conducted a search for the existing RC cars available with open-source simulators. In this section, we first list the set of hardware and software requirements to be met by the RC car projects. Next, we introduce the reader to the available open-source projects found during our search process, along with a short description for each of them. Finally, we motivate our choice of the DONKEY CAR as the reference architecture for our studies.

4.1 Requirements

The RC car's architecture should meet the following *hardware requirements*:

- **A remote control car body** which determines the maximum speed and steering angle, as well as all other driving parameters.
- **An on-board small-sized computer** mounted on top of the RC car, for the real-time DNN model's inference.
- **A camera sensor** which is the main input of every self-driving car system.
- **A powering system** capable of powering both the car's Electronic Speed Controller (ESC) and the on-board computer.

Additionally, the chosen architecture should also meet the following *software requirements*:

- **Availability of a Simulator** to conduct the experiments with a high-fidelity virtual model of the RC car.
- **Availability of a Large community** where issues are solved promptly and code and data are shared among members, which favors development.
- **High Maturity** since more mature projects typically imply more robust functionalities.

4.2 Results of the Search

Table 4.1 summarizes the main characteristics of the surveyed cars. The table reports the name of the project, the on-board computer architecture, information about the controller and the sensors, the availability of a simulator and an approximate cost in USD at the time of writing (May 2021). Next, we briefly describe the characteristics of each architecture.

Table 4.1. Comparison of RC car kits

Project	**Computer**	**Controller**	**Sensors**	**Simulator**	**Cost**
Donkey Car	RaspBerry/Jetson Nano	Web/Joystick	8MP camera	Y	250 - 350 $
DeepPiCar	RaspBerry	Web/UI	640x480 pixels camera	N	250 - 300 $
JetRacer	Jetson Nano	None	8MP camera	N	400 / 600 $
AWS DeepRacer	Proprietary	Web	4MP camera	Y	399 $
AWS DeepRacer Evo	Proprietary	Web	2x 4MP cameras, LIDAR	Y	599 $

4.2.1 Donkey Car

Donkey Car is an open-source do-it-yourself (DIY) self-driving platform for small-scale cars. The name derives from the fact that "*Donkey's are one of the earliest domesticated pack animals*", "*They are safe for kids*" and "*They occasionally don't follow their masters commands*".

The Donkey Car has only one sensor, a front-facing *8MP IMX219* camera which allows for a 160 degrees field of view (FoV) and up to 30 fps in 1080p quality recording. A car can be assembled using either a Raspberry Pi 3 or an NVIDIA's Jetson Nano computer. By the time our survey was conducted (August 2020), four supported versions of the Donkey Car were available. That is, documentation on how to assemble a car was provided for four different chassis all made under the "Exceed" brand: *Magnet*, *Desert Monster*, *Short Course Truck* and *Blaze*. The assembly part is well documented through online documentation and videos. Plastic-printed components can be bought online as well as the others components that are needed (e.g., computer, camera sensor, batteries, micro SD card).

A donkey car can be controlled through a physical gaming controller (e.g., Playstation 3, PlayStation 4, Xbox One, WiiU Pro, Logitech F710) or by using a virtual joystick provided on a web server hosted by the car's computer. The Donkey Car can be trained using the *Amazon Elastic Compute Cloud (Amazon EC2)* web service, which allows for cloud computations, or within a local host computer.

The Donkey Car project also comes with an ad-hoc simulator developed in Unity, which we can use to run experiments for online and offline testing on an additional environment.[1] Finally, we have found the Donkey Car community to be very active, providing regular updates and

[1] https://unity.com/

upgrades to the project via a Discord server, and organizing virtual competitions for hobbyists and enthusiasts.

4.2.2 DeepPiCar

DeepPiCar is an open-source project by David Tian [90]. The project allows to choose between three different RC car kits, while the on-board computer is a Raspberry Pi 3 B+. A Google Edge TPU accelerator is used for faster CNN inference.

The peculiar feature of this project is that the camera can be shifted both horizontally and vertically, potentially allowing to simulate the three cameras mounted on top of the improved Udacity simulator.[2] The main limitation of this project is that the default RC car only allows for three steering angles, i.e., the car has a very limited discrete action space, which does not quite match the real world where a car operates using a much larger and continuous action space. Moreover, we did not find much community activity in terms of project updates and datasets. Finally, the project does not come with a simulator.

4.2.3 JetRacer

JetRacer is another open-source project that gives a step-by-step guide on how to build a self-driving RC car. The project allows to pick between two different RC cars: the *Latrax Rally* and the more powerful *Tamiya TT02*.

The interesting features of the project are the use of the Jetson Nano computer, which allows for faster inference of the CNN, and the use of some very fast RC cars. On the other hand, this project features the most expensive kit and the documentation and community support does not seem to be as wide as DONKEY CAR's. Moreover, the project does not provide a simulator.

4.2.4 AWS DeepRacer

DeepRacer is Amazon's 1/18th scale race car. The car is mainly designed to test reinforcement learning (RL) models by racing on a physical track. By the time our study was conducted, Amazon offered two versions of the car: *DeepRacer* and *DeepRacer Evo*.

DeepRacer comes equipped with an Intel Atom Processor, a 4 MegaPixels camera, 4GB of RAM and 32GB of storage memory. Also, the car is provided with accelerometer and gyroscope sensors. The cost of the car is $399.00. *DeepRacer Evo* is an extension of the original car. A second stereo camera is added as well as a light detection and ranging (LIDAR) with a 360 degrees 12 meters scanning radius. This extension allows any model to use the additional information provided by the additional hardware. The sensor kit comes at a price of $249.00. When buying the standard DeepRacer with the sensor kit, the total cost sums up to $598.00.

Amazon offers a racing simulator that developers can use to collect training data and test their models on different tracks. However, users have a very limited set of DNN architectures that they can choose from and have to work with a RL framework.

Through the website, Amazon offers a complete documentation from the assembly to the model evaluation phases. Also, Amazon offers sample code and pre-trained RL agents to get quickly started. In spite of the fact that the car is mainly thought for RL algorithms, it should be possible to implement other types of model. This is however not documented at all in Amazon's website.

[2] https://github.com/udacity/self-driving-car-sim

4.2.5 Summary

We have overviewed a number of open-source projects available regarding self-driving RC cars, and all of them propose similar car builds and costs. However, only the DONKEY CAR project fulfills all our requirements, namely a simulator, support from the community and support for the supervised learning setting all at once. In fact, the DeepPiCar and JetRacer projects lack a simulator, while the AWS DeepRacer would restrict us to the usage of AWS technology. Moreover, AWS DeepRacer is designed mostly to study RL-based autonomous vehicles over supervised learning ones.

An own implementation of a self-driving RC car would in theory be possible, but such a solution is more expensive in terms of time with respect to an open-source project implementation. For these reasons, for the purpose of this thesis, we chose the DONKEY CAR project as our reference architecture.

Chapter 5

The Donkey Car Project

In this chapter, the reader is introduced to the Donkey Car project. In particular, we give an overview of the open-source project, a description of the physical RC car, the racing track used for our experiments, and the functionalities already implemented by the open-source community at the time of the studies.

5.1 Overview

The DONKEY CAR project allows developers to (1) build an electric RC self-driving car, (2) train different types of machine learning models both with real data or with simulated data, and (3) deploy them easily to control the RC car.

Each DONKEY CAR is composed of an RC car, a computer, and a camera sensor. The project's documentation guides the developers into assembling different configurations of these three elements.[1] The DONKEY CAR's project provides Python scripts to drive the car manually, to train DNN models and to deploy them. The user can select various parameters and options when running each script. For example, when collecting data, the user can specify whether she is using a joystick. When training a model, the user can specify which data shall be used, which model type should be trained (e.g., regression CNN, classification CNN, or RNN), whether to use data augmentation and more. Additionally, a configuration file allows to specify a number of parameters, such as:

- **Vehicle:** The rate at which the computer shall send signals to the ESC, run predictions and so on.
- **Camera:** Type of the camera, resolution, flipping and cropping of the image.
- **Steering:** Pulse Width Modulation (PWM) values to be sent to the ESC for maximum steering towards the left and right.
- **Throttle:** PWM values to be sent to the ESC for neutral throttle, maximum forward throttle and maximum reverse throttle.
- **Training:** Model training parameters such as model type, batch size, number of epochs, early stopping patience, optimizer, learning rate and more.

[1] Additional information at https://docs.donkeycar.com/guide/build_hardware/

- **Joystick:** Joystick deadzone (i.e. zone where moving the stick does not have any effect on the car) and maximum throttle percentage.

- **Leds:** color settings for signaling alerts, such as when a certain number of images has been captured.

- **Simulator:** Whether to drive the vehicle on the simulator or in the real world.

- **Others:** other options, such as values of p, i and d for a PID controller, or maximum throttling when an AI is being deployed.

The typical usage of the framework is as follows:

1. First, data is collected into driving episodes called *tubs*. To collect the training data one can either manually drive the Donkey Car through a web interface (using a joystick or a virtual trackpad) or, in the case of simulation, let a PID controller automatically drive the car around the track. This process results in a folder (tub) containing as many images as frames in the execution of the drive. Each image is associated with a `json` file containing information about the throttle and the steering angle that were observed when the frame was recorded. One can also collect multiple tubs and train a model using the entirety of the collected data.

2. Second, a model is trained with the use of a Keras/Tensorflow utility script. The scripts scans in the default tub folder which contains the previously collected tubs. Alternatively, the user can specify which tubs to use for training the model. Training parameters such as data split, epochs, early stopping criterion and more are specified in the configuration file. The architecture of the model can be selected out of 8 default models provided by the project.

3. Finally, the newly trained model is evaluated on its capability of controlling the vehicle. An option in the configuration file allows to record the driving performance of the trained model within a tub.

Note that steps one and three can be carried out either in “the real world” or in simulation. The tub is obtained each time the Donkey Car moves, either in autonomous or manual mode. Apart from these functionalities, the project also features virtual and physical competitions which are organized by the Donkey Car enthusiasts and hobbyists. Furthermore, we have found the Donkey Car community to be very active, as we have witnessed many updates to the project while this thesis was being conducted.

5.2 The Car

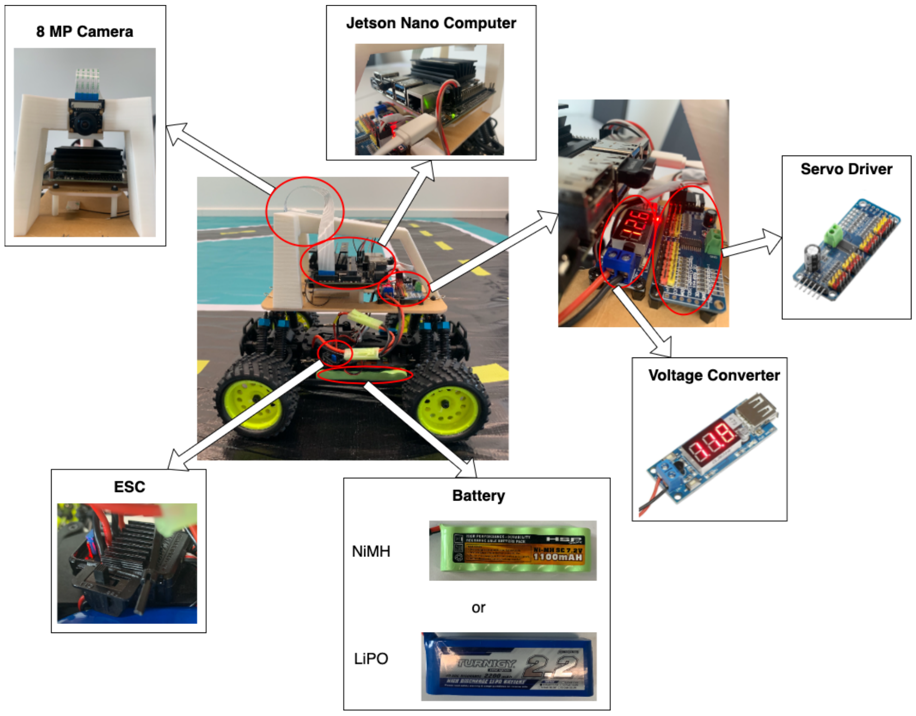


Figure 5.1. Schematic overview of our own DonkeyCar

For this study, the standard starter kit *Jetson Nano Edition* of the DONKEY CAR, provided by the *Robocarstore* was used.[2] The kit comes with a **HSP 94186 Brushed RC Car**, a **Jetson Nano** computer and a **Sony IMX219 camera**, along with other components needed to assembly and connect the three main components.

Figure 5.1 shows a schematic overview of our own DonkeyCar. A LiPO battery powers the Electronic Speed Controller and, through the voltage converter, the Jetson Nano computer which senses the environment through the 8MP camera sensor and sends commands to the ESC via the PCA 9685 servo driver.

5.2.1 RC Car

The HSP 94186 Brushed RC Car is a 1/16 RC car featuring a 2040 Brushless Motor and a 7.2V 1100mAh Ni-MH SG Battery for a driving time without recharge of approximately 15 minutes. The Electronic Speed Controller (ESC) controls the car steering and throttle.

5.2.2 Computer

NVIDIA's Jetson Nano is a small yet powerful computer optimized for real-time neural networks predictions. Jetson Nano features a 128-core Maxwell GPU, a Quad-core ARM A57 1.43GHz

[2] https://www.robocarstore.com/products/donkey-car-starter-kit-jetson-nano-edition

CPU, 4GB of LPDDR4 RAM and an NVIDIA Tegra X1 GPU which allows it to handle tasks, among the others, of image classification, object detection and segmentation and it is thus a suitable computer for the RC car. The size of the Jetson Nano computer is 70mm x 45mm.

5.2.3 Camera

The IMX219-160 Camera is an 8 MegaPixels camera which features 160 degrees of Field-of-view (FOV) and a maximum resolution of 3280 $\times$ 2464 pixels. The Lens is characterized by a CMOS size of 1/4 inch, an aperture of 2.35 (F), a focal length of 3.15mm, a diagonal angle of view of 160 degrees and distortion that is under 14.3%. The camera size is 25mm x 24mm. The camera is compatible with NVIDIA's Jetson Nano.

5.2.4 Battery

The battery provided in the starter kit has a capacity of 7.2V (1100mAH). Despite being fully functional, we found that this battery does not allow for extensive online testing of models as model inference on the Jetson Nano is a power-hungry task. Also, quick battery discharges have effects on the driving capability of the DONKEY CAR which slows down as the battery charge is consumed. For this reason, we replaced the default battery with an 11.3V 2.2A LiPo battery that allows for longer and more consistent online testing.

5.2.5 Other components

The other components provided in the *Robocarstore* bundle are the following:

- **Intel Wifi card and antennas**: Allow the computer to connect to Wi-Fi and bluetooth connections. This is necessary as a wireless connection to the car's computer is key to remotely start and stop the vehicle.

- **Top Cage**: The top cage is a white 3D-printed component used to fix the camera on top of the car. Also, the shape is designed such that the car can be grabbed from the top.

- **Servo Driver PCA 9685**: An Inter-Integrated Circuit (I2C). This component connects the computer to the RC car's steering and throttle servo motors.

- **DC-DC 5V/2A Voltage Converter**: Connects the RC car battery to the computer and provides an indication of the voltage supplied by the battery through a LED indicator.

- **32GB MicroSD card**: Constitutes the non-volatile memory of the Jetson Nano on which the operating system is installed and records of the sessions are stored, as well as the models and other files.

- **Base Plate**: It's a laser-cut base. Most of the components are mounted on top of this base, namely the computer, the top cage, the servo driver and the voltage converter.

- **Wires and screws**: Used to connect and fix all components on top of the base plate.

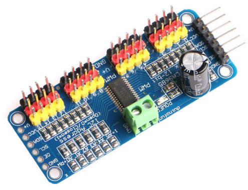
5.2.6 Assembly

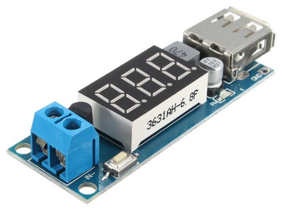
Although the assembly of this particular edition of the Donkey Car was not documented neither in the *Robocarstore*, nor in the Donkey Car official website, we found it to be straightforward. The car ESC is connected to the servo driver which is connected to the computer, such that the computer can send the throttle and steering signals. The car battery is connected both to the car ESC and to the voltage converter which is itself connected to the Jetson Nano, such that both the computer and the ESC can be powered by the same battery. The IMX219 Camera is plugged into the Jetson Nano's MIPI CSI-2 DPHY lane and then attached to the white top cage, which is finally put on top of the car.

(a) Jetson Nano

(b) RC Car

(c) IMX219 Camera


(d) PCA9685 Servo Controller


(e) Voltage Converter

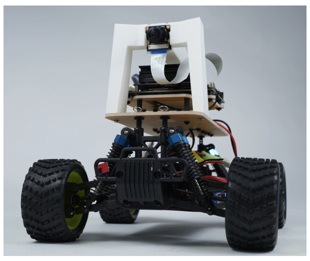

(f) Fully assembled car

Figure 5.2. Our Donkey Car's main components

5.3 The Racing Track

We performed our experiments on the default racing track provided by the Robocar Store website, called DIYRobocars Standard Track. The track is printed on top of a plastic mat and is a standard track used in the DONKEY CAR community. In fact, different physical competitions are based on this track and it also resembles the generated track scene provided in the DONKEY CAR simulator. This track is available in four scales: 100%, 80%, 60% and 40%. In our studies, we have used the 40% scale version of the track, which weights 5 kilograms and has size 3.00m x 4.54m. This track is both used in the simulator and real world. Because of the steering capabilities of the RC car used in our studies, the tightest turn in the track cannot be traversed without exceeding the track bounds. This, however, does not represent a limit for the study.

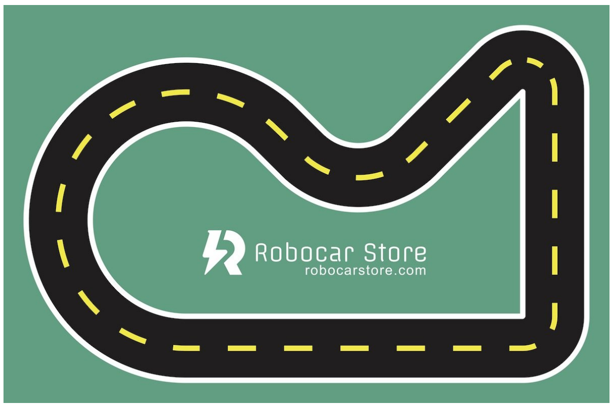


Figure 5.3. Robocar Store default track

5.4 Donkey Environment

In order run the scripts, a conda environment named “donkey” is provided. The conda environment comes with all libraries and frameworks needed. Those include Tensorflow Keras, NumPy, OpenCV, OpenAI’s Gym and more.[3][4][5][6]

The DONKEY CAR environment also offers a variety of utility commands such as:

- **calibrate**: Allows to calibrate car steering and throttle prior to racing.
- **tubclean**: Allows to manually clean data in a tub in a simple interactive way. By cleaning data in a tub, one typically removes bad driving samples such as, for example, when the car was driven off road.
- **makemovie**: Creates an `mp4` video file given a tub of camera frames. Interestingly, an option also allows to show the saliency map with the pixels that mostly activated the neural network.

[3] https://keras.io/
[4] https://numpy.org/
[5] https://opencv.org/
[6] https://gym.openai.com/

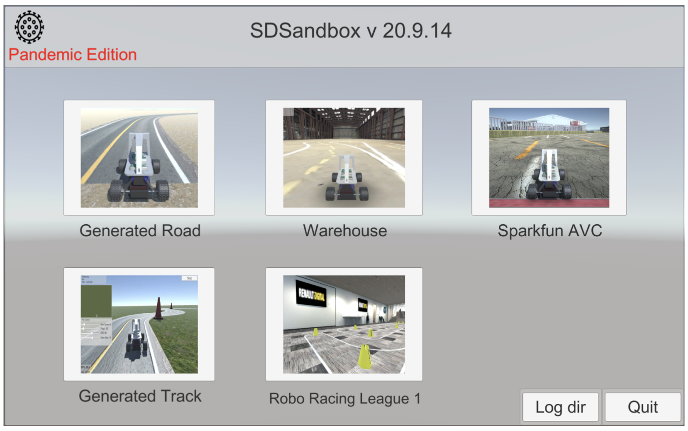


Figure 5.4. Simulator's main menu (September 2020)

- **tubaugment**: Allows to automatically augment data contained in a tub applying horizontal flipping, saturation and other transformations to the images in the tub.
- **cnnactivations**: Shows the features maps for each filter in the convolutional layers.

5.5 Donkey Simulator

The Donkey simulator (self-driving Sandbox or "sdsandbox") is an open-source Unity project to train and test models for the DONKEY CAR in a virtual environment in a supervised way.[7] The DONKEY CAR community uses the simulator to compete in online events such as the *Virtual Race League*.[8] The Unity project is available for Windows, MacOS and Linux operating systems. For our studies, we modified the original sdsandbox project by implementing a customized track which resembles our real world scene. Further details are available in Chapter 6.

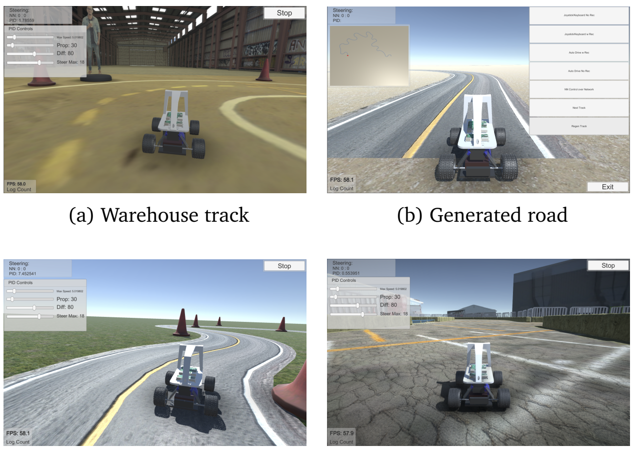
5.5.1 Main Simulated Tracks

For our studies, we used version 20.9.14 of the sdsandbox project, which provides four defaults tracks. For each track scene the following commands are possible when the scene is loaded:

- **Drive manually**: Allows the user to drive the car in the track with a joystick / keyboard.
- **Autodrive**: A PID controller drives the car autonomously around the track by using waypoints.
- **Neural Network Drive**: Deploys a machine learning model onto the car which will decide steering angle and throttle of the car.

In all modes, runs can be recorded within tubs.

[7] https://docs.donkeycar.com/guide/simulator/
[8] https://docs.donkeycar.com/guide/virtual_race_league/



(a) Warehouse track (b) Generated road

(c) Generated track (d) Sparkfun AVC track

Figure 5.5. Donkey simulator's main tracks

Warehouse track

The warehouse track (see figure 5.5a) is a scene that depicts a warehouse in which a small track is drawn on top of the warehouse floor. This scene is a virtual version of a really existing warehouse track in Oakland (United States) where developers used to gather data and race in monthly events. The scene features the warehouse, a track drawn on the warehouse floor, some red cones and static silhouettes representing the developers.

Generated road

The generated road (see figure 5.5b) allows the user to generate pseudo-random roads through two additional commands: *Regen Track* and *Next Track*. The former command keeps the track unchanged, but alters the texture. The latter clears the current road and creates a new random one with a new texture and width. The generated roads are built based on newly created waypoints, thus it is possible to use a PID controller to automatically gather training data. Also, generating new roads randomly allows to easily extend the training and testing corpus.

Generated track

The generated track (see figure 5.5c) is a track that is generated "on the fly" like in the Generated road scene, but the exact track configuration is specified in a text file as a scripted path. The scene also involves a static grass texture and 13 red cones around the border of the track.

Sparkfun AVC

Sparkfun AVC (see figure 5.5d) is another track inspired from a past competition (now retired): the Sparkfun's Autonomous Vehicle Competition in Boulder, Colorado (United States). The competition first took place in 2009, and took place every year up until 2018. The track is an eight-shaped road delimited by hay bales, featuring barrels, bumpy ramps, a stop sign and a pedestrian silhouette. The scene also features, in the background, buildings, cars silhouettes, gazebos and others.

5.5.2 PID Controller for automated data collection

The dataset needed to train and test a model, both in simulated and real environments, is typically collected by an expert driver who manually drives the vehicle. With each camera frame, information about the throttle and steering given by the driver in that particular moment in time is stored, and the model is trained to imitate the expert driver's behaviour.

The DONKEY CAR simulator, however, can also make use of PID controllers that automatically drive the car on a road by following waypoints. This feature is particularly interesting because it allows to collect training data without the need of an expert (human driver), and can even yield better results as it iss much less likely for the PID controller to drive off-road.

Mathematical model of a PID Controller

A PID controller (Proportional-Integral-Derivative controller) is a control loop that allows to adjust some controllable variables in an iterative way such that a goal behaviour is achieved (see Figure 5.6). A PID controller uses an error measurement to adjust the controllable variables. In particular, the adjustment is taken based on part of the current error (proportional), on the cumulative error over time (integral) and on the difference between the current error and the error measured in the previous time-step (derivative).

Typically, the error e(t) is computed as a difference between a goal output of the system and the current output value. The mathematical model is the following, let:

- **t** be the current time-step
- **e(t)** be the error measured at time *t*
- $\mathbf{K_p}$ be the proportional coefficient (constant)
- $\mathbf{K_i}$ be the integral coefficient (constant)
- $\mathbf{K_d}$ be the derivative coefficient (constant)
- **u(t)** be the control given at time *t*

Then, at each time-step, the control that tries to minimize the error for the next time-step is obtained as:

$$u(t) = K_p * e(t) + K_i \int_0^t e(t')dt' + K_d \frac{de(t)}{dt}$$

In other words, the next action to be taken by the system is a linear combination of the current error, the cumulative error witnessed so far and the difference between the last two witnessed errors. Note that constants K_p, K_i and K_d are all non-negative constants which have to be tuned.

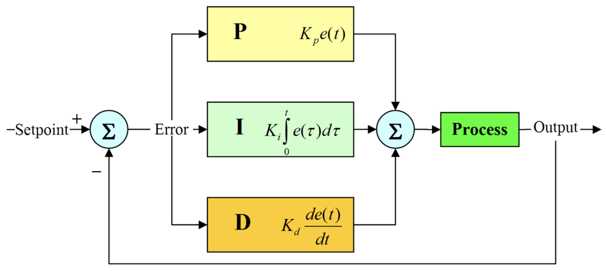


Figure 5.6. Illustration of a PID Controller

The Cross Track Error

The PID controller needs some error measure $e(t)$ with each time step. This measure is called cross-track error (CTE or XTE) and it is obtained through some hidden waypoints on the track.

The XTE is computed as follows: Let C be the 3-dimensional position of the center of mass of the car. Let S be the segment connecting the active waypoint to the next one. Now let P be the closest point to C lying on S. The XTE is the magnitude (norm) of the vector going from P to C. Note that the sign of e(t) is adjusted depending if the car is "to the left" of segment S, in which case it would be negative, or "to the right", in which case it would be positive.

The XTE thus basically penalizes deviations from the line connecting two consecutive waypoints in a linear manner. Note that waypoints are always placed in the center of the track, such that the absolute value of the cross track error provides information about how much the car is deviating from the center trajectory. Relevant waypoints to be considered "current" are updated automatically by the simulator.

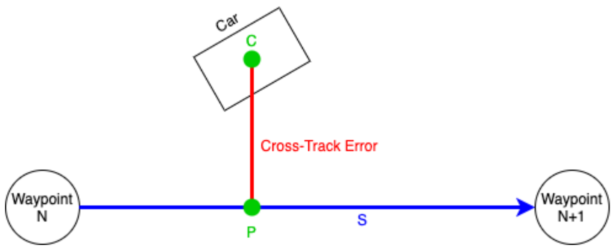


Figure 5.7. Schematic view of the cross-track error

Figure 5.7 shows a representation of how the cross track error is obtained. The segment S (blue line) connects the active waypoint (N-th waypoint) to the next one. Point C (green) represents the center of mass of the car, while point P (green) is the closes point to C that lies on S. The magnitude of the vector connecting point P and C represents the absolute value of the cross track error. The sign of the cross track error is determined by which side the car is located with respect to segment S.

In the simulator, the values of K_p and K_d are adjustable with a slider when selecting the "Autodrive" option. Constants K_p, K_i and K_d are set to 30, 0 and 80 respectively by default. Moreover, in the DONKEY CAR project, waypoints are implemented in the "PathNode" class. Each PathNode has a *pos* attribute which specifies the 3-dimensional position in the simulator.

5.5.3 Automatic Simulated Track Generation with Waypoints

The Donkey simulator allows to automatically build tracks with hidden waypoints given a specification of the track layout in a configuration (text) file. The configuration of the track can be given in two formats: as a “scripted path” or as a “point path”.

Scripted paths

The file that defines a scripted path is composed of lines of the form “COMMAND ARGS”, where COMMAND is a string and ARGS is an integer or real number.

The main available commands are:

- **S**: stands for “Straight”. When this command is selected, the the simulator builds a straight road segment as long as the specified amount in the parameter.
- **L/R**: stand for “Left” and “Right” respectively. The simulator builds a left or right bend road segment by the specified amount.
- **DY**: Specifies how tight the curves should be from this point onward in the track. A higher value tells that curves should be tighter, while a lower value tell that curves should be wider.

For example, the instructions: “DY 1”, “L 10”, “DY 10”, “R 10” specify a road that slightly goes to the left and then has a tighter right curve. This example is presented in Figure 5.8.

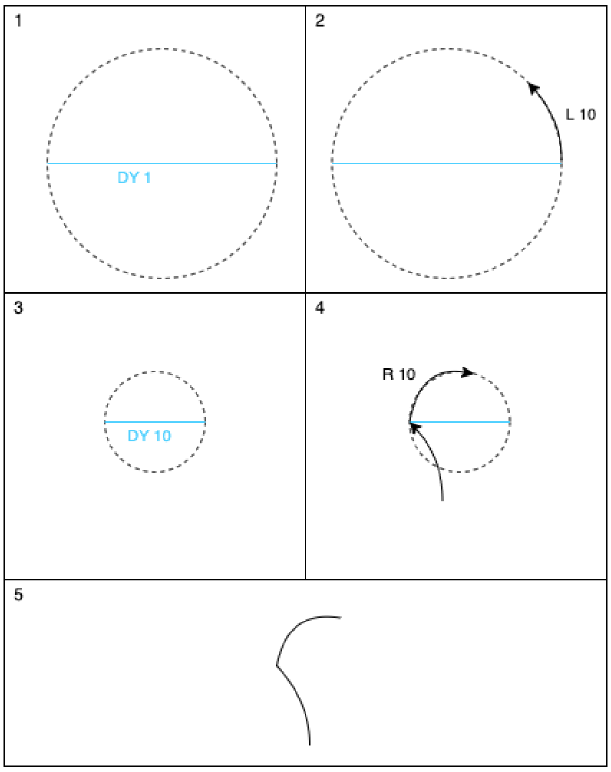


Figure 5.8. Visual representation of the scripted path function

Point paths

The most straightforward way to define a track consists of manually defining the waypoints in the 3-dimensional space. In this configuration, a comma-separated values file (csv) is used, where each line defines the position of the next waypoint with three real number values representing the (x, y, z) coordinates.

This option allows great flexibility and control in terms of generated tracks, as higher waypoint granularity allows a more accurate approximation of the road's shape, especially for bends. However, specifying a track manually is a much more tedious and challenging task compared to the scripted path approach. In fact, the DONKEY CAR project provides examples of both files to reconstruct the "warehouse" track. The scripted version of the track can be defined in a 16 lines long file with 66 characters, while the point path version of the same track is 174 lines long with almost 5'000 characters.

Chapter 6

Simulated Testing Environment

Having simulated and real testing environments that look alike is an essential requirement for our studies. Hence, we developed a new Unity scene within the original simulator that highly resembles the actual room in which we executed our experiments in the real world. Having similar simulated and real environments does not only allows comparing online and offline testing within the same environment, but also across different environments. Another additional benefit is the possibility to test whether a simulator is representative of the dynamics occurring in the real world.

In this chapter, we present how we built our simulated environment. Specifically, we describe the ad-hoc Unity scene that was built, the adjustments and improvements we have done to the DONKEY CAR project and, finally, the dataset we have collected for our simulated models.

6.1 Customized Scene

The simulated scene was designed to be as similar as possible to the real world room that we used for testing. In order to do so, proportions of the track and of the car with respect to the track have been maintained according to the original tracing track and RC car. We called such scene "*USI track*".

6.1.1 Track

We used a high-resolution picture of the real track and we have imported it within our Unity scene. This ensures that the track used in the simulator has exactly the road shape and colours as the one printed onto the plastic carpet in the real world (Figure 5.3).

The image is used as a texture over a 2-dimensional plane in Unity. Approximately, the track's width is twice as large as the RC car. We adjusted the proportions of the plane (height and width) such that the proportions of the car with respect to the track are the same as the proportions of the real world.

As introduced in Chapter 5, a PID controller can be used on a track that features waypoints. We manually placed 22 waypoints onto the track. We then use an ad-hoc script that finds the 3-dimensional position of the waypoints in the simulator and writes them on a text file. The PID controller script, when executed, reads the text file and the position of the car from the simulator and emits new commands for the car. Our simulated track is thus suited for automatic

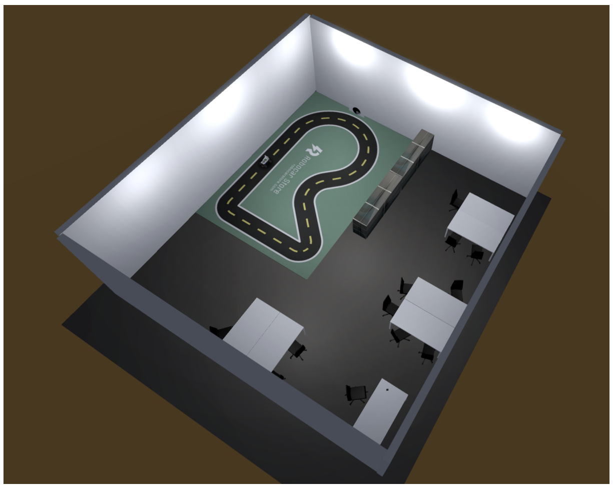


Figure 6.1. Simulated environment

data collection, since the PID controller can be used to traverse the track while staying as close as possible to the center of the road.

6.1.2 Simulated Vehicle's Camera

The camera mounted on top of the simulated car was made as similar as possible to the real one by adjusting the camera parameters in Unity. Those parameters are the camera's focal length, which we set to 3.15mm, and sensor size, set to 16mm.

6.1.3 Room

Figure 6.1 shows our simulated environment Unity scene from a top-down view. We emulated the lightning present in the real environment by placing nine spotlights on the virtual ceiling. The lights are disposed according to their position in the real environment. Other objects, such as chairs and tables are present in the scene to simulate the non-relevant information that might be present into the camera feed.

6.2 Extensions to the Donkey Car Project

During our studies, we also modified parts of the source code of the original Donkey Car project, which are described next.

6.2.1 Donkey camera resolution

By default, the frames from the car's camera are collected with a fixed resolution. However, different model architectures were designed with different input sizes. For example, DAVE-2 was designed to process 66x200 pixels input images, whereas the original Chauffeur model uses inputs of size 120x320. We modified the source code of the simulator such that the desired car's camera resolution can be selected manually by the user through a drop-down menu from the main menu scene. When the virtual car runs, the collected images are stored in the selected resolution.

For our studies, we set the camera resolution to 320x240 pixels for all driving models to ensure uniformity across experiments.

6.2.2 Computation of Waypoints

The simulator handles the processing of the next waypoint that needs to be traversed by the PID controller. However, for waypoints that are very close to each other, a major bug was observed. Particularly for our track, the default PID controller script was not able to finish an entire lap around the track. This was due to the placement of one waypoint in the proximity of the narrowest curve. If the car's travel trajectory was skipping the waypoint in that curve, the waypoint was not considered as passed, or driven through. This caused erroneous calculations of the XTE metric that we fixed by modifying the way in which a waypoint is considered passed. More specifically, when the angle between the car's orientation vector and the vector connecting the car's center of mass to the waypoint is bigger than 90 degrees, the waypoint is considered as passed and the next waypoint is considered the current one.

6.2.3 Logging of Telemetry

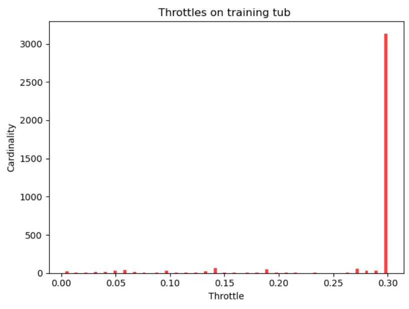
Each time the car is driven in the simulator, either in manual or autonomous mode, a tub can be recorded. When a tub is recorded, all camera frames that the virtual car's camera processed are stored, along with the steering and throttle commands applied by the driver or predicted by the DNN model.

We extended the logging system to also report: the XTE, lap number, maximum XTE during this lap so far, number of times the car has driven out of bounds, the car speed and acceleration. The information about the XTE is taken by a dedicated script, developed in the context of previous research [84]. The lap number is implemented by counting how many times the car passes through the first waypoint. We count how many times the car went off road by counting the number of times the XTE exceeds 1.5 in absolute value. Finally, the Unity physics engine allows to retrieve velocity and acceleration of a physical object automatically.

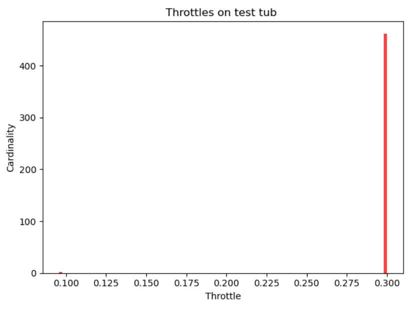
6.3 Dataset

We have collected a training and a testing set of 320x140 RGB images, each annotated with the respective throttle and steering angle used to traverse the track. Both training and test set were collected from the custom version of the Donkey car simulator which features our own "*USI track*" scene, logging of the XTE, adjustable camera resolutions (of the car) and even domain randomization (i.e. automatic altering of wall colors, light positions and light colors).

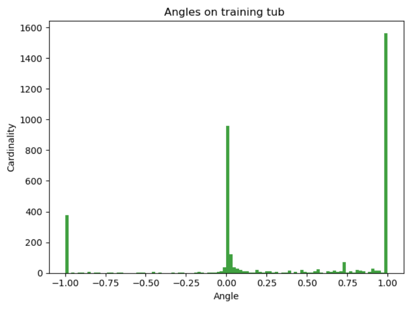
The *training set* was collected by driving the car with a controller, collecting 3'657 images over 10 laps. Note that with respect to the real world, models can learn to successfully drive in the simulated environment using a smaller dataset. This is probably justified by the fact that the real world is richer in detail, and images vary much more than in the simulated environment. Although the PID controller could have been used to generate data, we eventually decided to collect the dataset by driving manually. This choice is motivated by the fact that manual driving (1) yields a dataset that better resembles the one collected in the real world, including recovery



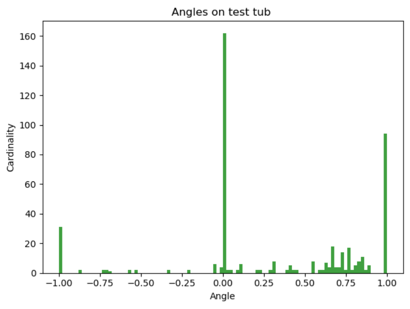
(a) Distribution of throttle in train set



(b) Distribution of throttle in test set



(c) Distribution of steering angles in train set



(d) Distribution of steering angles in test set

Figure 6.2. Throttle and angle distributions in training and test set for the simulated testing environment

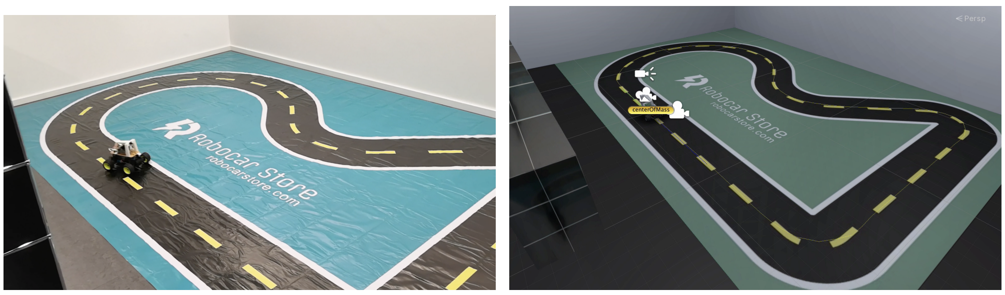
episodes and (2) a relatively small amount of data is needed to train models that drive in the simulator. Similarly, we have manually collected a single-lap *test set* driving the car with a trajectory that, approximately, minimizes the overall cross track error. The test set is composed of 464 images.

Figure 6.2 shows the distribution of throttle (red) and angle (green) for the training and test sets in the simulated environment. We can observe that the label distributions in the training and test set are quite similar. In fact, the one dominant value for throttle is 0.3, i.e., 30% of the maximum throttle for both training and test sets. We have limited the throttle of the simulated car to 30% to be more controllable by a human driver. Similarly, the three main dominant values for the steering angle in both the training and test sets are -1, 0 and 1 (meaning steering all left, driving straight and steering all right).

Chapter 7

Real world Testing Environment

In our study, we are interested in learning differences and analogies between offline and online testing. To do so, we study these two typologies of testing both in a real world and simulated environment. In this chapter, we describe the dataset that was collected from the real world environment. This dataset will be used to train multiple models in Chapter 8. We then also describe an approach based on CycleGAN for estimating the cross track error given a real world image frame.



(a) Real environment (b) Simulated environment

Figure 7.1. Pictures of the real and simulated environments

Figure 7.1 shows a view of the real (left) and simulated (right) environments.

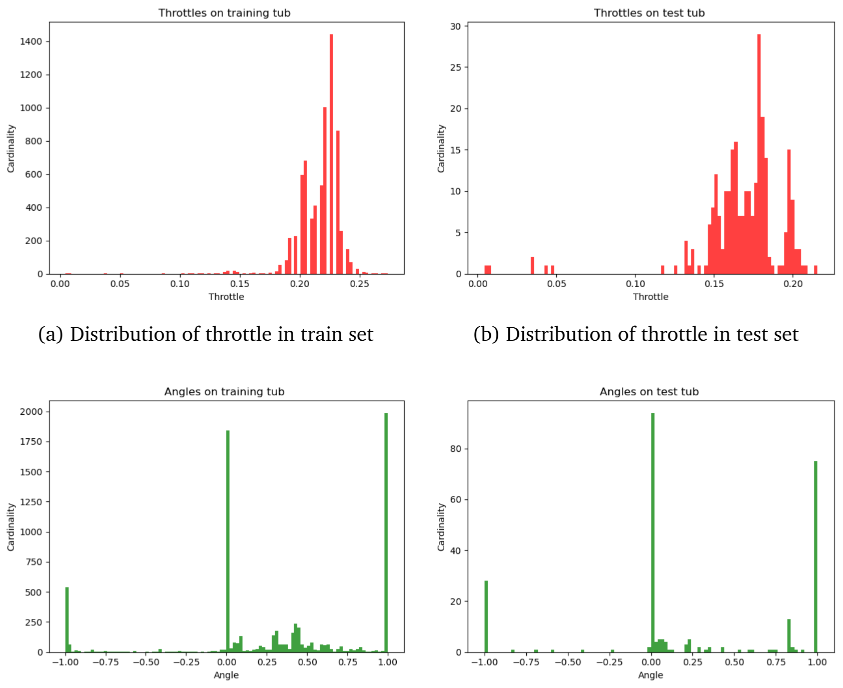
7.1 Dataset

7.1.1 Training set

In order to train models that can drive on the real track, we collected a dataset by manually driving the Donkey Car with a PlayStation 4 controller. The training set consists of a 7 minutes clockwise drive around the track, featuring 7'079 annotated RGB images of size 320x240 over 33 laps.

7.1.2 Test set

Similarly as for the training set, a one-lap test set was collected manually with the use of a PlayStation 4 controller. The test set is composed of 262 RGB images of size 320x240 taken over a single lap.



(a) Distribution of throttle in train set (b) Distribution of throttle in test set

(c) Distribution of steering angles in train set (d) Distribution of steering angles in test set

Figure 7.2. Throttle and angle distribution in training and test set for the real environment

In Figure 7.2 we observe the distribution of labels, i.e., steering angles (green) and throttles (red), in both the training and test set for the real environment.

The plots concerning the distribution of throttle (in red) show that values are mostly concentrated between 0.2 and 0.25 for the training set and between 0.15 and 0.2 for the test set. Note that these values represent a percentage with respect the maximum throttle that the RC car can deliver. In fact, driving the car any faster than this becomes extremely challenging as reaction times for the human shorten. Note that the values of throttles slightly differ in training and test sets as, when collecting the test set, we paid much attention in driving the car as close as possible to the center of the road, while sacrificing the driving speed. This, however, does not represent a problem as we are mainly interested in the steering angle prediction errors.

The plots concerning the distribution of steering angles (green) instead highlight how three main angles are adopted by a human driver: -1, 0 and 1, which mean all the way left, center and all the way right respectively. This distribution can be justified by the fact that the steering capabilities of the car are not sufficient to effectively traverse the track without using the maximum and minimum steering angles. Also, given the relatively small scale of the track, it is far easier for a human driver using a PlayStation 4 controller to zig-zag through the track using

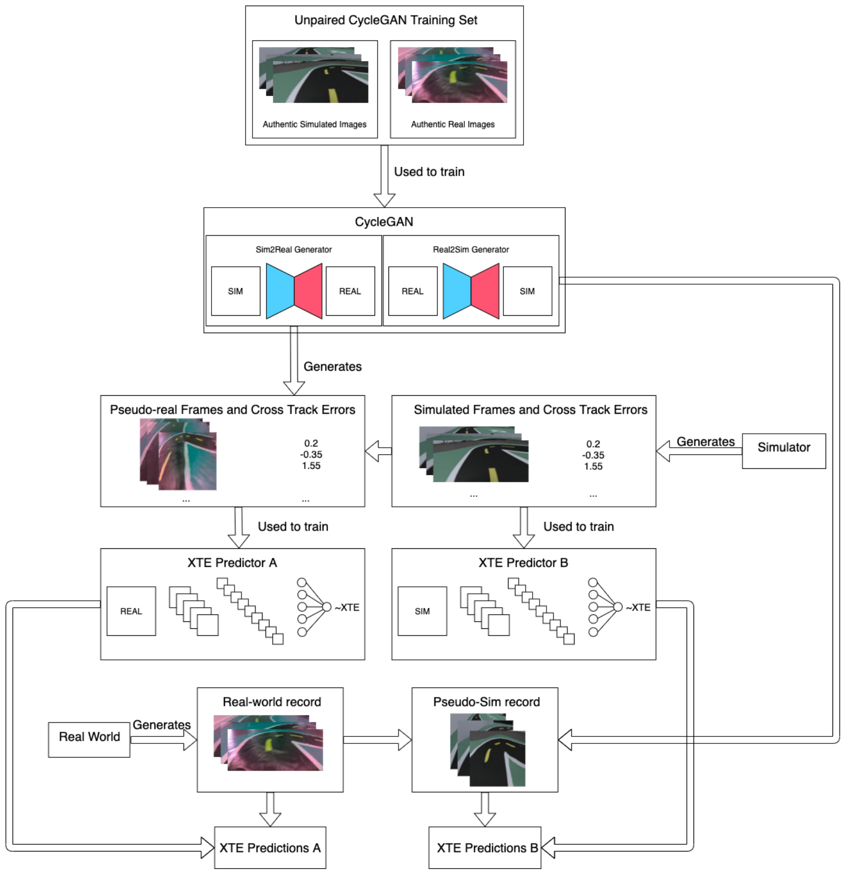


Figure 7.3. Schematic view of how the XTE is estimated for the real world

maximum/minimum steering rather than continuously and slightly adjusting the trajectory at each instant in time.

7.2 Real world cross track error estimation

In the simulator, for each frame recorded by the car we store the associated cross track error, which measures how far away from the center of the track the center of mass of the car is. However, in the real world, such information is not readily available from any of the car's sensors. This is because we cannot locate a point in the middle of the road that is closest to the center of mass of the car. Also, in the general case, it is challenging to locate a moving object in an indoor space with high accuracy [47].

We can, nonetheless, estimate the cross track error by exploiting the only piece of information we are given, that is, the camera feed. We refer to our estimate of the cross track error as approximate XTE (~XTE). Our approach for estimating the XTE is based on DL. Although more traditional computer vision algorithms might be applicable for our particular setting, they would not transfer to different scenarios and non-fixed environments. We believe that our DL approach, instead, can be used for any self-driving system and generalize to more environments. Estimating the XTE for the real world is beneficial as it constitutes an important metric for online and simulated testing [48, 37, 17].

Figure 7.3 presents a schematic view of the two different routes we have to estimate the XTE. An unpaired dataset of images from both distributions is used to train translators using CycleGAN. The XTE for a real image can be estimated either by (a) using a XTE predictor trained on pseudo-real data paired with simulator labels or (b) translating the real world image to a pseudo-simulated image and using a XTE predictor trained on original simulated data.

7.2.1 Methodology

Let X be the probability distribution from which real world images are sampled, and Y the probability distribution from which the simulator images are sampled, we can train a Cycle-GAN [103] model to translate images from the simulator into real images (sim-to-real) using function f and vice-versa (real-to-sim) using function g previously discussed in Chapter 3. We call simulated images that are translated into real images “pseudo-real”, as they come, indeed, from a generative process. Similarly, we call real world images that are translated into simulated images “pseudo-simulated”.

To estimate the real-world XTE, we propose two approaches.

First approach: sim2real

After training a CycleGAN model for making such translations, we collect a simulated tub of N images, where for each training image we have a corresponding XTE accurately computed by the simulator. Mathematically, we describe such dataset as:

$$\Theta_{sim} = \{(I_{sim}^{(i)}, e^{(i)})\} \quad \forall i \in [1, \dots, N]$$

Where $I_{sim}^{(i)}$ is the i-th image coming from the simulator, and $e^{(i)}$ is the i-th relative cross track error recorded at that particular frame. We can obtain a pseudo-real version of the dataset by translating all simulator-generated images using function f:

$$\Theta_{pseudo-real} = \{(f(I_{sim}^{(i)}), e^{(i)})\} \quad \forall i \in [1, \dots, N]$$

The newly generated dataset can now be used to train a CNN for the task of regression of the XTE. The main assumptions are as follows: if the pseudo-real generated images are representative enough of real real world images, and if the CNN learns to generalize well enough, then the predictor will yield accurate approximations also for real images. To estimate the real world XTE for an image I_{real}, we can thus use this XTE predictor which we call XTE predictor A, and denote mathematically as Φ_A. We thus have that our estimation will be $\Phi_A(I_{real})$.

Second approach: real2sim

This XTE predictor would be trained on the original simulated dataset Θ_{sim}. We refer to this predictor as XTE predictor B, which we denote mathematically as Φ_B. When we need to estimate real world XTE, we can then translate the real world image into a pseudo-simulated image which will then be run through the XTE predictor B. The XTE estimate is thus $\Phi_B(g(I_{real}))$.

Comparison of approaches

We have presented two ways to estimate the real world cross track error. With the first approach, we learn a predictor Φ_A trained on pseudo-real data. This is then used to make inferences on some real world data. With the second approach instead, we learn a predictor Φ_B trained on simulated data. We then use Φ_B to make inferences on pseudo-simulated data. Naturally, the performance of the predictor are dependent on the capabilities of the generative model to reproduce inputs across distributions (real and simulated track in our setting) with high fidelity (or with a low error).

Our CycleGAN model

We trained our CycleGAN model using 5’360 simulated images and 6’639 images taken from our real environment. The RGB images were collected with an original size of 320x240 pixels, which we then cropped to size 320x120 to remove background noise and reshaped to a size of 256x256 pixels. The output of the CycleGAN model is a new 256x256 pixels image. The network was trained for 50 epochs with the standard CycleGAN learning rate of 0.0002, and for further 20 epochs featuring a linear decay of the learning rate towards zero. The generator architecture used is a 9-blocks Resnet [40], while the discriminator architecture is the default CycleGAN discriminator architecture.

Our XTE predictors models

We use the same architecture for predictors Φ_A and Φ_B. The architecture is the Donkey Car default CNN model presented in Figure 3.2, except that the input shape of the model is 256x256. This is because our CycleGAN model outputs images of this dimensions. We train such models using the Adam Optimizer [53] with a learning rate of 10^{-5}, optimizing the MSE. We use 90% of data for training, and validate on 10% of the data using a 10-epochs patience early stopping.

Quality assessment of generated images

We assessed the quality of the generated images empirically, since it is an important factor for obtaining a good XTE predictor. We collected two recordings both from the simulator and from the real environment. We tried to align the recordings such that at each frame the car is located in a similar position across records, and videos have the same amount of frames. Then, we translated simulated images into pseudo-real images using function f and real world images into pseudo-simulated images using function g. Finally, we computed the Frechét Inception Distance (FID) [42], a metric typically computed for comparing images generated with GANs against original images using the activation traces of an Inception network [87] trained for the ILSVRC dataset [55], between the authentic and artificial images for both the environments.

We obtain two FID scores: one between real world images and pseudo-real images, and one between simulated images and pseudo-simulated images.

Since the original simulated and real world tubs are not perfectly aligned, we expect the FID scores to be a pessimistic estimate of the generator model capabilities. That is, if we could collect perfectly aligned original tubs, we would be able to observe lower FID scores. More importantly, the Inception network used to compute the FID was trained on the ImageNet dataset, which does not feature images that resemble tracks.

We can, nonetheless, compare the values of the FID scores obtained between real and pseudo-real images and between simulated and pseudo-simulated images.

Distributions	**FID**
Real & Pseudo-Real	**106.8**
Sim & Pseudo-Sim	158.1

Table 7.1. Fréchet inception distance for the two distributions (original and artificial) of both environments

Table 7.1 shows the FID scores obtained between real with pseudo-real and simulated with pseudo-simulated images. We can notice that the FID is lower for real and pseudo-real images. In other words, function f does a better job at translating simulated images into pseudo-real images than function g does at translating real world images into pseudo-simulated images.

One reason for this, could be the imbalance in our CycleGAN training set. In fact, more images of the real-world were used to train the CycleGAN model, while less images of the simulated environment were used.

Overall, this result is positive, as typically one is interested in generating high fidelity images of the real world and not of a simulated environment.

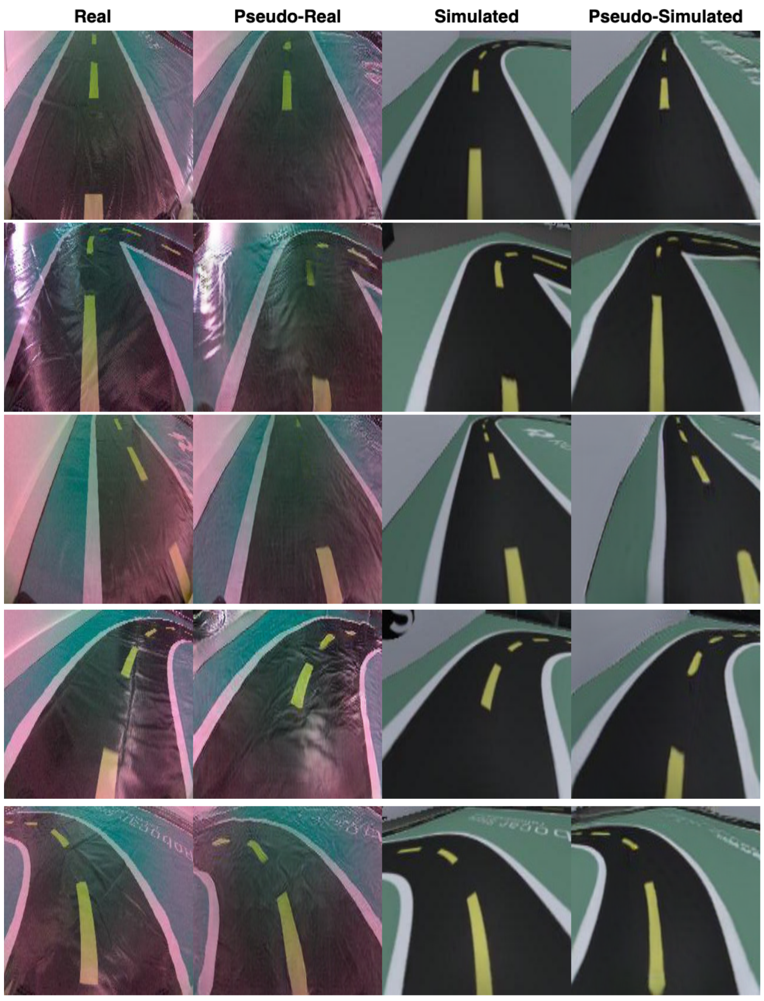


Figure 7.4. Examples of original and artificially generated images

In Figure 7.4, we show some examples of original and artificial images for both environments. Notice that pseudo-real images (second column) are obtained using function f on simulated images (third column). Similarly, pseudo-simulated images (fourth column) are obtained using function g on real images (first column).

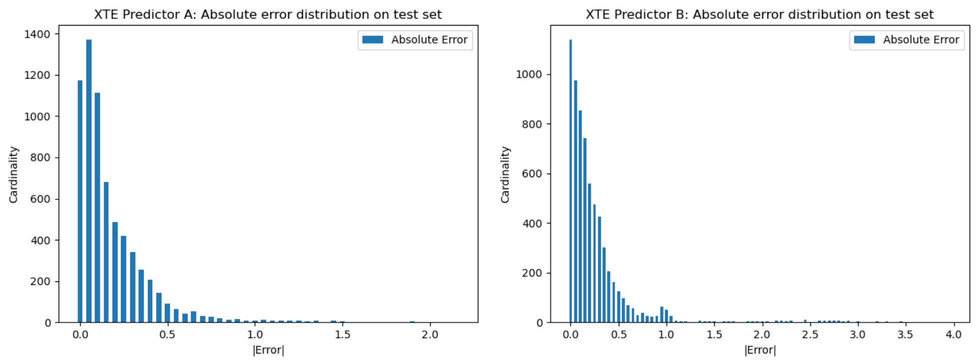
7.2.2 Evaluation of the XTE predictors

To validate our XTE predictors, we manually collected a test set for the task of regression of the XTE. In order to do so, we estimated how far from the center of the track the car should be in the real world to have an absolute XTE of 1. This is done by visualizing the position of the car in the simulator, and replicating such position in the real world. We found that an absolute XTE of 1 corresponds to 11.7 cm in the real world.

After doing so, we collected a total of 6'638 test images from the real environment using the Donkey Car, where we estimated the XTE to be -2 (1'463 images), -1 (1'528 images), 0

(1’663 images), 1 (1’114 images) and 2 (870 images). Note that a cross track error of 0 means that the vehicle is exactly in the middle of the track. A cross track error of -1 or +1, or in other words 11.7 cm away from the center of the road, corresponds to the car barely touching the white border of the track with both wheels facing forward. A cross track error of -2 and +2, or 23.4 cm away from the center of the road, means that the center of mass of the car is exactly on top of the white border of the track.

To collect images where the XTE was -2, -1, 0, 1 and 2, we have driven one slow lap around the track by keeping the car on the left white border, middle of the left lane, middle of the track, middle of the right lane and on the right white border respectively. Collecting test images where the XTEs vary from -2 to +2 means spanning the main regions of interest. Actually, this test set does not represent a ground truth, as such ground truth cannot be measured exactly. However, we believe that the test set we have collected is in practice as good as any test set that can be produced manually, and a model that performs well on such test set can be considered a good estimator.



(a) Absolute error distribution of XTE Predictor A (b) Absolute error distribution of XTE Predictor B

Figure 7.5. Distribution of the absolute error of our XTE predictors against our test set.

We tested our XTE predictors on the manually collected test set to determine which model yields the best predictions. Notice that, for XTE predictor B, real world images were translated into pseudo-simulated images using CycleGAN.

Figure 7.5 shows the absolute errors of our XTE predictors on our test set. The figure highlights how most of the absolute errors made by our cross track predictor A are low. In fact, half of the absolute errors is lower than 0.134 (around 1.57cm) and, on average, the error is lower than 0.2 (around 2.34cm) for predictor A. Predictor B turned out to be less effective, as it features a higher absolute error mode (0.173) and mean (0.282).

We also run the Wilcoxon-signed-rank test [73] using the differences between predictions obtained with Φ_A and Φ_B. We obtained a p-value of 0.0774, indicating that our XTE predictors likely differ in the corresponding prediction medians.

Additionally, we compute Cohen’s d [18] between predictions of Φ_A and Φ_B on our test set. We found d to be 0.020673, hinting that the effect size is small.

The superiority of Φ_A over Φ_B on the test set might be due to the ability of our CycleGAN model to better translate simulated images into real-images, and thus providing a better training set for Φ_A. To improve performances of Φ_B, we suspect it would be necessary to train a CycleGAN model on an evenly balanced dataset featuring as many images of the real world as images from the simulator.

For our online testing experiments, we will only use our Φ_A estimator, as it is arguably the best estimator, in our particular case, among the two we have proposed.

Chapter 8

Experiments and Results

In this chapter, we discuss the empirical study that we carried out to answer our research questions. In each experiment, we compare online with offline testing both in the simulator and the real world.

In our first experiment, we use the default DONKEY CAR model with three different values of cropping for the model's input image to empirically select the optimal input size. In our second experiment, we compare the different self-driving car model architectures described in Chapter 3, namely, DAVE-2, Epoch, Chauffeur and the default DONKEY CAR model.

8.1 Metrics

Here, we describe the metrics used for evaluating a model both with offline and online testing. For each model tested with offline testing, we report the steering and throttle MSE and MAE on the 1-lap test set in both environments.

For each model tested through online testing, we report the average and maximum XTE or ~XTE, the steering mean and standard deviation, the throttle mean and standard deviation and the number of laps concluded successfully out of 10. Notice that the ~XTE is obtained using Φ_A, as it yielded better performances on the test set.

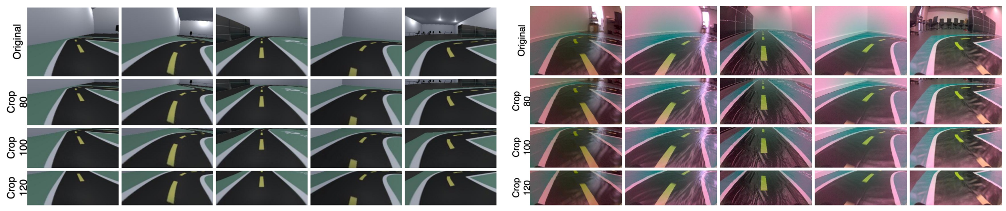
We do not report the number of times the car exited the track or crashes, as we found these statistics to be the same for all models in both environments.

8.2 Cropping Experiment

In the DONKEY CAR project, it is possible not only to specify the car's input image size, but also the vertical cropping. In fact, one can specify the amount of top and bottom pixels to be removed before passing the input image to the model.

Cropping is very useful for the model mounted on the Donkey car as the model is not fed with useless information such as background and noise and only focuses on the relevant features (i.e., the track) yielding better results with less computations. However, cropping relevant information could yield less accurate inference and might result in sub-optimal behavior.

In this experiment, we set the car's camera image size to 320x240 pixels and train the same DONKEY CAR default model three times with three different values of vertical cropping from the top, namely 80, 100 and 120 pixels of crop. All other training parameters were fixed, namely architecture, optimizer, learning rate, data augmentation, number of epochs and early stopping patience (in epochs). We call the models *Crop80*, *Crop100* and *Crop120*, respectively. The models were tested both offline and online and both in the simulator and in the real world.



(a) Cropping in simulated environment (b) Cropping in real environment

Figure 8.1. Cropping in the simulated and real environment

Figure 8.1 shows the original image (320x240 pixels) and the three versions of cropping both for the real environment and the simulated environment. Note that in both environments (1) The original image features a lot of noise, as the top 320x100 pixels always include just background. (2) The 80 pixels crop image removes most of the background but leaves some of it. (3) The 100 pixels crop seems to be the optimal cropping value, as it discards background information and keeps the visible track. (4) The 120 pixels crop deletes useful information from the image, in some cases even hiding the next imminent turn ahead.

8.2.1 Results for Cropping Experiment

Table 8.1 shows the results for the *offline testing* both in the simulator and in the real world respectively. The table reports the model's name and size in number of trainable parameters, and the MSE and MAE values for steering and throttle for both simulated and real world testing environments. We can notice that, for offline testing, the *Crop100* model achieves the best results on both environments. For the simulated environment, the *Crop80* model achieves comparable results as the *Crop100* model. The *Crop120* model shows lower effectiveness for both environments by a discrete margin.

		SIMULATED				REAL WORLD			
		Steering		**Throttle**		**Steering**		**Throttle**	
Model	**Param.**	MSE	MAE	MSE	MAE	MSE	MAE	MSE	MAE
Crop80	2'897'028	0.071	0.153	0.002	0.037	0.094	0.195	**0.003**	**0.044**
Crop100	2'263'428	**0.068**	**0.151**	**0.001**	**0.021**	**0.070**	**0.163**	0.004	0.056
Crop120	1'841'028	0.078	0.163	**0.001**	0.022	0.117	0.204	**0.003**	0.046

Table 8.1. Offline testing for simulated and real world testing environments for the cropping experiment (best results are highlighted using boldface).

Table 8.2 shows the results for online testing. The table reports the average and max XTE, the steering and throttle values and the number of laps. The 'Train set' row of the tables shows

Model	Avg XTE	Max XTE	Steering	Throttle	Laps
SIMULATED					
Train set	*0.569*	*3.437*	*0.614 ± 0.457*	*0.278 ± 0.063*	*10*
Crop80	**0.459**	3.894	0.578 ± 0.357	0.258 ± 0.036	**10**
Crop100	0.498	**3.636**	0.578 ± **0.356**	0.275 ± **0.027**	**10**
Crop120	0.514	3.224	0.487 ± 0.374	0.284 ± 0.030	1
REAL WORLD					
Train set	*0.762*	*2.980*	*0.533 ± 0.406*	*0.193 ± 0.012*	*30*
Crop80	**0.979**	**2.770**	0.463 ± 0.352	0.243 ± **0.008**	**10**
Crop100	1.043	2.772	0.540 ± **0.340**	0.238 ± 0.010	**10**
Crop120	N/A	N/A	N/A	N/A	0

Table 8.2. Online testing for simulated and real world testing environments for the cropping experiment (best results are highlighted using boldface).

the human performance when collecting the training data on which the models were trained. The remaining rows display the results for the different models.

Also for online testing, the model *Crop120* is inferior to the models *Crop80* and *Crop100*, as it only drove a single lap in the simulation environment and no lap in the real world before going out of track irremediably. Thus, online and offline testing are consistent with respect to the worst performance of the *Crop120* model. On the other hand, online testing for the real world shows that the *Crop80* seems to be slightly superior to *Crop100*, while for the simulated environment we conclude that they are pretty much equivalent.

8.3 Architectures Experiment

In the second experiment, after tuning the optimal input size, we study the effectiveness of the self-driving car model architectures under test: DAVE-2, Chauffeur and Epoch (Chapter 3).

		SIMULATED				REAL WORLD			
		Steering		Throttle		Steering		Throttle	
Model	Param.	MSE	MAE	MSE	MAE	MSE	MAE	MSE	MAE
Chauffeur	63’818	**0.074**	**0.151**	0.002	0.026	0.097	0.228	**0.002**	**0.030**
DAVE-2	2’249’030	0.098	0.173	0.003	0.040	0.108	0.219	0.004	0.061
Default	2’263’428	0.089	0.157	0.002	0.025	**0.070**	**0.163**	0.004	0.056
Epoch	22’307’362	0.091	0.164	**0.001**	**0.020**	0.106	0.210	0.003	0.047

Table 8.3. Offline testing for simulated and real world testing environments for the architectures experiment (best results are highlighted using boldface).

Table 8.3 shows the results for the *offline testing* both in the simulator and in the real world respectively. The table reports the model’s name and size in number of trainable parameters, and the MSE and MAE values for steering and throttle for both simulated and real world testing

Model	Avg XTE	Max XTE	Steering	Throttle	Laps
SIMULATED					
Train set	*0.569*	*3.437*	*0.614 ± 0.457*	*0.278 ± 0.063*	*10*
Chauffeur	0.508	2.720	0.601 ± **0.351**	0.299 ± 0.047	**10**
DAVE-2	0.536	3.000	0.617 ± 0.395	0.278 ± 0.058	**10**
Default	**0.496**	3.113	0.578 ± 0.356	0.275 ± **0.027**	**10**
Epoch	0.577	**2.532**	0.479 ± 0.423	0.313 ± 0.043	**10**
REAL WORLD					
Train set	*0.762*	*2.980*	*0.533 ± 0.406*	*0.193 ± 0.012*	*30*
Chauffeur	1.358	2.851	0.522 ± **0.321**	0.236 ± 0.014	**10**
DAVE-2	**0.613**	**2.223**	0.712 ± 0.392	0.202 ± **0.010**	**10**
Default	1.043	2.772	0.540 ± 0.340	0.238 ± **0.010**	**10**
Epoch	0.671	2.435	0.437 ± 0.326	0.217 ± 0.017	**10**

Table 8.4. Online testing for simulated and real world testing environments for the architectures experiment (best results are highlighted using boldface).

environments. Unlike in the previous experiment, where offline testing was yielding same conclusions across environments, here offline testing finds better suited models for each individual environment. In particular, for the simulated environment, Chauffeur is the best model, followed by the Default DONKEY CAR model, Epoch and finally DAVE-2. For the real environment, however, this ranking is not maintained. In fact, for the real world environment, the model that performs best according to offline testing is the DONKEY CAR default model. These results hint that the two environments, from the perspective of ML models, differ substantially, despite the high resemblance observed visually.

Table 8.4 shows the online testing metrics for the different models in the simulated and real environment, respectively. We can notice that online testing in different environments yields different outcomes. This once more hints that the two environments, although visually similar for a visual perspective, yield very different driving performances for SDC models.

For offline testing in the simulated environment, the Chauffeur model is the one that better generalizes to unseen road images with respect to the others by a good margin (Table 8.3). Online testing however does not clearly identify a best driving model, as models with lower average XTE had higher maximum XTE and vice-versa (Table 8.4).

Similarly for the real environment, offline testing predicted that the default DONKEY CAR model was superior to the others. The Avg XTE measured in the real world is constantly higher, whereas the Max XTE is comparable.

8.4 Answers to the Research Questions

8.4.1 Answer to Research Question One

How do offline testing metrics in a simulated environment compare to the same metrics in a real environment?

While in our first experiment differences in offline testing metrics across environments were not very pronounced (see Table 8.1), in our second experiment (see Table 8.3), where input images share the same resolution for all models, we can see that these vary importantly. This result suggests that visual resemblance between environments, as perceived by humans, does not translate to similar behaviours of ML models across environments.

8.4.2 Answer to Research Question Two

How do online testing metrics in a simulated environment compare to the same metrics in a real environment?

Online testing metrics differ importantly across environments. Some models seem to perform better in our simulated environment, while others yield better driving behaviors in the real world. Once again we are hinted that our simulator, despite generating images that resemble the real world scene, is not a good “proxy” of the real world. In other words, we could not learn from the simulator what would be the model that shows the best driving behaviour in the real world.

8.4.3 Answer to Research Question Three

Can offline testing replace online testing?

The answer to this question is no. Through our empirical approach, we have found that offline testing does not correctly reveal which model yields the best driving performances when deployed onto the vehicle. This is especially true for our second experiment, where we train multiple architectures on a fixed dataset. Furthermore, we find that there is little correlation between offline and online testing, as models that perform relatively poorly in offline testing (with respect to other models) do not necessarily perform poorly when deployed onto the car. Online testing is thus definitively a necessary step in the development of a SDC.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this thesis, we compared online and offline testing of deep neural networks for autonomous vehicles both in a simulated and real environment.

To do so, we relied on the DONKEY CAR open-source project and modified the existing simulator by generating a custom Unity scene that resembles the real world and by logging important online testing metrics such as XTE, mean and standard deviation of steering, velocity, acceleration and others. We also proposed an approach based on CycleGAN [103] to estimate the XTE on the real world given the car's camera frame.

We then trained and tested different models both in the simulator and in the real world both in the online and offline settings. We collected the results and, based on these, we have drawn our conclusions and answers to the research questions.

We ultimately concluded that our simulated environment, despite being specifically designed to resemble as much as possible our real environment, is not a good representative for telling how model architectures trained on simulated data might perform when trained for the real world. We have found that models with a higher number of trainable parameters behave better in the real world, hinting that the simulated environment might be too simplistic and unable to capture the variability of the real world.

Finally, we concluded that offline testing cannot replace online testing, as little correlation between the results we obtained for both environments was found.

9.2 Future Work

In this section, we propose some ideas that could represent the next steps towards testing of SDCs and improvements to our current work.

9.2.1 Training Self-Driving Models with Pseudo-Real Data

In Chapter 7, we proposed an approach for estimating the XTE in the real world through CycleGAN [103]. Our method uses CycleGAN to learn a function that can translate real images into pseudo-simulated images and a function that translates simulated data into pseudo-real images.

For future work, pseudo-real images could be used to train ML models in the task of driving. In particular, one could collect a dataset of simulated images and relative driving commands, translate the simulated images into pseudo-real images and use these, jointly with the driving commands labels, to train a model to be deployed on the SDC. This idea follows the intuition that since images generated through CycleGAN look photo-realistic, they can be used as training data.

If successful, this approach could be used to collect images that actually come from the simulator, but look as if they were drawn from a real world distribution. This would allow to collect photo-realistic images typically hard to collect in the real world, such as dangerous driving scenarios, variable weather conditions and more. This data could finally be used to train safer and more robust ML models for SDCs.

9.2.2 Online Testing with Pseudo-Real Environments

Using CycleGAN, one could translate the images recorded by the car's camera into another distribution in real time. We could translate images recorded in our testing lab into images that look as if they were taken from a urban city, a mountain road, a desert highway and more. This would result in a multitude of possible scenarios on the same physical track, which would allow for more exhaustive online testing in the real world.

An ML model capable of driving in different artificial scenarios would be regarded as more robust with respect to models that fail in doing so. The fundamental assumption of this idea is that the GAN model is able to correctly reconstruct the position of the road while also generating highly realistic images.

9.2.3 Reinforcement Learning

Typical reinforcement learning algorithms use, as reward function for SDCs models, the distance travelled by the car before going off road, or other intrinsic reward functions [3, 51, 54].

Our estimate of the XTE, however, could constitute an explicit reward signal. It would thus be possible to use this estimate in a reward function for an RL agent such that the goal of the agent would then be to minimize the ∼XTE. This estimate can be particularly beneficial for RL algorithms, as the agents could learn to stay as close as possible to the center of the lane, rather than just avoiding exiting it.

9.2.4 Improvements to the Simulator

In our studies we have shown that our simulator, despite recreating the real-world scene quite well for a human observer, is not representative enough of the real scene for machine learning models to behave similarly to what they would do in the real world.

One extension of our work would be to improve the current simulator to generate photo-realistic images. This could be achieved through ray-tracing, a computer graphics technique that simulates real world lighting, combined with accurate emulation of the materials of which objects in the scene are composed.

It would then be possible to repeat the experiments that were carried out in our simulator and measure whether, and by which margin, a more sophisticated simulator reduces the gap between offline and online testing results across environments.

Bibliography

[1] Eric Adams. Why we're still years away from having self-driving cars, 2020. URL https://www.vox.com/recode/2020/9/25/21456421/why-self-driving-cars-autonomous-still-years-away.

[2] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017. doi: 10.1109/ICEngTechnol.2017.8308186.

[3] A. Amini, I. Gilitschenski, J. Phillips, J. Moseyko, R. Banerjee, S. Karaman, and D. Rus. Learning robust control policies for end-to-end autonomous driving from data-driven simulation. *IEEE Robotics and Automation Letters*, 5(2):1143–1150, 2020. doi: 10.1109/LRA.2020.2966414.

[4] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017. doi: 10.1109/TPAMI.2016.2644615.

[5] Reinhold Behringer, S. Sundareswaran, Billy Gregory, R. Elsley, B. Addison, Wayne Guthmiller, R. Daily, and D. Bevly. The darpa grand challenge - development of an autonomous vehicle. pages 226 – 231, 07 2004. ISBN 0-7803-8310-9. doi: 10.1109/IVS.2004.1336386.

[6] Keshav Bimbraw. Autonomous cars: Past, present and future - a review of the developments in the last century, the present scenario and the expected future of autonomous vehicle technology. *ICINCO 2015 - 12th International Conference on Informatics in Control, Automation and Robotics, Proceedings*, 1:191–198, 01 2015. doi: 10.5220/0005540501910198.

[7] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars, 2016.

[8] Mariusz Bojarski, Chenyi Chen, Joyjit Daw, Alperen Değirmenci, Joya Deri, Bernhard Firner, Beat Flepp, Sachin Gogri, Jesse Hong, Lawrence Jackel, Zhenhua Jia, BJ Lee, Bo Liu, Fei Liu, Urs Muller, Samuel Payne, Nischal Kota Nagendra Prasad, Artem Provodin, John Roach, Timur Rvachov, Neha Tadimeti, Jesper van Engelen, Haiguang Wen, Eric Yang, and Zongyi Yang. The nvidia pilotnet experiments, 2020.

[9] Houssem Ben Braiek and Foutse Khomh. On testing machine learning programs. *Journal of Systems and Software*, 164:110542, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2020.110542. URL https://www.sciencedirect.com/science/article/pii/S0164121220300248.

[10] Zhaowei Cai, Quanfu Fan, Rogerio S. Feris, and Nuno Vasconcelos. A unified multi-scale deep convolutional neural network for fast object detection. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 354–370, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46493-0.

[11] Oliver Cameron. We're building an open source self-driving car. Technical report, 2016. URL https://medium.com/udacity/were-building-an-open-source-self-driving-car-ac3e973cd163#.xvv50px6m.

[12] Oliver Cameron. Teaching a machine to steer a car. Technical report, 2016. URL https://medium.com/udacity/teaching-a-machine-to-steer-a-car-d73217f2492c.

[13] Davide Castelvecchi. The black box of ai. *Nature*, 538:20–23, 2016. doi: 10.1038/538020a.

[14] Kaihu Chen. Image operations with cgan, 2021. URL http://www.k4ai.com/imageops/index.html.

[15] HeeSun Choi, Cindy Crump, Christian Duriez, Asher Elmquist, Gregory Hager, David Han, Frank Hearl, Jessica Hodgins, Abhinandan Jain, Frederick Leve, Chen Li, Franziska Meier, Dan Negrut, Ludovic Righetti, Alberto Rodriguez, Jie Tan, and Jeff Trinkle. On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward. *Proceedings of the National Academy of Sciences*, 118(1), 2021. ISSN 0027-8424. doi: 10.1073/pnas.1907856118. URL https://www.pnas.org/content/118/1/e1907856118.

[16] Dan Claudiu Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Twenty-second international joint conference on artificial intelligence*, 2011.

[17] Felipe Codevilla, Antonio M. López, Vladlen Koltun, and Alexey Dosovitskiy. On offline evaluation of vision-based driving models, 2018.

[18] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.

[19] DARPA. The darpa grand challenge: Ten years later, 2014. URL https://www.darpa.mil/news-events/2014-03-13.

[20] Leonid Datta. A survey on activation functions and their relation with xavier and he normal initialization, 2020.

[21] Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for machine learning*. Cambridge University Press, 2020.

[22] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[23] Li Deng and Dong Yu. Deep learning: Methods and applications. *Found. Trends Signal Process.*, 7(3–4):197–387, June 2014. ISSN 1932-8346. doi: 10.1561/2000000039. URL https://doi.org/10.1561/2000000039.

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

[25] Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia. Verifai: A toolkit for the design and analysis of artificial intelligence-based systems, 2019.

[26] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2018.

[27] Fenglei Fan, Jinjun Xiong, and Ge Wang. On interpretability of artificial neural networks. *CoRR*, abs/2001.02522, 2020. URL http://arxiv.org/abs/2001.02522.

[28] Daniel J. Fremont, Edward Kim, Yash Vardhan Pant, Sanjit A. Seshia, Atul Acharya, Xantha Bruso, Paul Wells, Steve Lemke, Qiang Lu, and Shalin Mehta. Formal scenario-based testing of autonomous vehicles: From simulation to the real world, 2020.

[29] Claudio Gambella, Bissan Ghaddar, and Joe Naoum-Sawaya. Optimization problems for machine learning: A survey. *European Journal of Operational Research*, 290(3):807–828, May 2021. ISSN 0377-2217. doi: 10.1016/j.ejor.2020.08.045. URL http://dx.doi.org/10.1016/j.ejor.2020.08.045.

[30] Alessio Gambi, Marc Mueller, and Gordon Fraser. Automatically testing self-driving cars with search-based procedural content generation. pages 318–328, 07 2019. doi: 10.1145/3293882.3330566.

[31] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[32] Hossein Gholamalinezhad and Hossein Khosravi. Pooling methods in deep neural networks, a review, 2020.

[33] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[34] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015.

[35] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, Apr 2020. ISSN 1556-4967. doi: 10.1002/rob.21918. URL http://dx.doi.org/10.1002/rob.21918.

[36] John K Haas. A history of the unity game engine. 2014.

[37] Fitash Ul Haq, Donghwan Shin, Shiva Nejati, and Lionel Briand. Comparing offline and online testing of deep neural networks: An autonomous car case study, 2019.

[38] Fitash Ul Haq, Donghwan Shin, Shiva Nejati, and Lionel Briand. Can offline testing of deep neural networks replace their online testing?, 2021.

[39] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[41] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[42] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium, 2018.

[43] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

[44] SAE International. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. page 35, 2018. doi: https://doi.org/10.4271/J3016_201806.

[45] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[46] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks, 2018.

[47] Yuya Itagaki, Akimasa Suzuki, and Taketoshi Iyota. Indoor positioning for moving objects using a hardware device with spread spectrum ultrasonic waves. In *2012 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–6, 2012. doi: 10.1109/IPIN.2012.6418850.

[48] Gunel Jahangirova, Andrea Stocco, and Paolo Tonella. Quality metrics and oracles for autonomous vehicles testing. 04 2021.

[49] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution, 2016.

[50] Andrej Karpathy. Ai for full self-driving [conference presentation]. 2020. Presented at the 5th Annual Scaled Machine Learning Conference 2020.

[51] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. Learning to drive in a day, 2018.

[52] Nikhil Ketkar. *Stochastic Gradient Descent*, pages 113–132. Apress, Berkeley, CA, 2017. ISBN 978-1-4842-2766-4. doi: 10.1007/978-1-4842-2766-4_8. URL https://doi.org/10.1007/978-1-4842-2766-4_8.

[53] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[54] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey, 2021.

[55] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012. doi: 10.1145/3065386.

[56] Kimin Lee, Honglak Lee, Kibok Lee, and Jinwoo Shin. Training confidence-calibrated classifiers for detecting out-of-distribution samples, 2018.

[57] W. Li, C. W. Pan, R. Zhang, J. P. Ren, Y. X. Ma, J. Fang, F. L. Yan, Q. C. Geng, X. Y. Huang, H. J. Gong, W. W. Xu, G. P. Wang, D. Manocha, and R. G. Yang. Aads: Augmented autonomous driving simulation using data-driven algorithms. *Science Robotics*, 4(28), 2019. doi: 10.1126/scirobotics.aaw0863. URL https://robotics.sciencemag.org/content/4/28/eaaw0863.

[58] Ming-Yu Liu, Thomas Breuel, and Jan Kautz. Unsupervised image-to-image translation networks, 2018.

[59] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets, 2014.

[60] Maryam M Najafabadi, Flavio Villanustre, Taghi M Khoshgoftaar, Naeem Seliya, Randall Wald, and Edin Muharemagic. Deep learning applications and challenges in big data analytics. 2(1), 2015. doi: 10.1561/2000000039.

[61] Nina Narodytska and Shiva Kasiviswanathan. Simple black-box adversarial attacks on deep neural netowrks, 2017.

[62] Kamyar Nazeri, Eric Ng, and Mehran Ebrahimi. Image colorization using generative adversarial networks. In Francisco José Perales and Josef Kittler, editors, *Articulated Motion and Deformable Objects*, pages 85–94, Cham, 2018. Springer International Publishing. ISBN 978-3-319-94544-6.

[63] Julia Nitsch, Masha Itkina, Ransalu Senanayake, Juan Nieto, Max Schmidt, Roland Siegwart, Mykel J. Kochenderfer, and Cesar Cadena. Out-of-distribution detection for automotive perception, 2020.

[64] NVIDIA. Virtual-based safety testing for self-driving cars from nvidia drive sim., 2021. URL https://www.nvidia.com/en-us/self-driving-cars/simulation/.

[65] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning, 2018.

[66] Union of Concerned Scientists. Maximizing the benefits of self-driving vehicles: Principles for public policy. Technical report, Union of Concerned Scientists, 2017. URL http://www.jstor.org/stable/resrep17299.

[67] Union of Concerned Scientists. Self-driving cars explained: How do self-driving cars work—and what do they mean for the future?, 2017. URL https://www.ucsusa.org/resources/self-driving-cars-101.

[68] Sean O'Kane. How tesla and waymo are tackling a major problem for self-driving cars: Data. Technical report, 2018. URL https://www.theverge.com/transportation/2018/4/19/17204044/tesla-waymo-self-driving-car-data-simulation.

[69] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[70] Dean A. Pomerleau. Alvinn: an autonomous land vehicle in a neural network, 1989.

[71] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. A survey on deep learning: Algorithms, techniques, and applications. *ACM Comput. Surv.*, 51(5), September 2018. ISSN 0360-0300. doi: 10.1145/3234150. URL https://doi.org/10.1145/3234150.

[72] Aishwarya Rathore. State-of-the-art self driving cars: Comprehensive review. 01 2016.

[73] Denise Rey and Markus Neuhäuser. *Wilcoxon-Signed-Rank Test*, pages 1658–1659. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-04898-2. doi: 10.1007/978-3-642-04898-2_616. URL https://doi.org/10.1007/978-3-642-04898-2_616.

[74] Vincenzo Riccio and Paolo Tonella. Model-based exploration of the frontier of behaviours for deep learning system testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 876–888, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409730. URL https://doi.org/10.1145/3368089.3409730.

[75] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. Testing machine learning based systems: A systematic mapping. *Empirical Software Engineering*, 11 2020. doi: 10.1007/s10664-020-09881-0.

[76] Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[77] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, Eugene Agafonov, Tae Hyung Kim, Eric Sterner, Keunhae Ushiroda, Michael Reyes, Dmitry Zelenkovsky, and Seonman Kim. Lgsvl simulator: A high fidelity simulator for autonomous driving, 2020.

[78] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[79] Bria Rosenberg. The benefits of self-driving cars. Technical report, VIA Technologies, Inc., 2019. URL https://www.viatech.com/en/2019/06/the-benefits-of-self-driving-cars/.

[80] 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.

[81] Juergen Schmidhuber. Prof. schmidhuber's highlights of robot car history. Technical report, 2011. URL https://people.idsia.ch/~juergen/robotcars.html.

[82] Salman Sherin, Muhammad Uzair khan, and Muhammad Zohaib Iqbal. A systematic mapping study on testing of machine learning programs, 2019.

[83] Kakul Shrivastava, Maruthi S Inukonda, and Sparsh Mittal. Hardware-software stack for an rc car for testing autonomous driving algorithms. 2019. doi: 10.13140/RG.2.2.14768.30728.

[84] Andrea Stocco and Paolo Tonella. Towards anomaly detectors that learn continuously. In *Proceedings of 31st International Symposium on Software Reliability Engineering Workshops*, ISSREW 2020. IEEE, 2020.

[85] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. Misbehaviour prediction for autonomous driving systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 359–371, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380353. URL https://doi.org/10.1145/3377811.3380353.

[86] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. A survey of optimization methods from a machine learning perspective, 2019.

[87] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

[88] TÜV SÜV. Autonomous vehicle, highly automated driving, 2021. URL https://www.tuvsud.com/en/industries/mobility-and-automotive/automotive-and-oem/autonomous-driving.

[89] Team Epoch. Steering angle model: Epoch. https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/cg23, 2016. Online; accessed 18 August 2019.

[90] David Tian. Deeppicar — part 1: How to build a deep learning, self driving robotic car on a shoestring budget. Technical report, 2019. URL https://towardsdatascience.com/deeppicar-part-1-102e03c83f2c.

[91] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars, 2018.

[92] Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christopher Bregler. Efficient object localization using convolutional networks, 2015.

[93] 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, 2018.

[94] Inc. Udacity. Become a self-driving car engineer. Technical report, 2021. URL https://www.udacity.com/course/self-driving-car-engineer-nanodegree--nd013.

[95] Yuri Viazovetskyi, Vladimir Ivashkin, and Evgeny Kashin. Stylegan2 distillation for feed-forward image manipulation, 2020.

[96] Ari Viitala, Rinu Boney, Yi Zhao, Alexander Ilin, and Juho Kannala. Learning to drive (l2d) as a low-cost benchmark for real-world reinforcement learning, 2020.

[97] Per Welander, Simon Karlsson, and Anders Eklund. Generative adversarial networks for image-to-image translation on multi-contrast mr images - a comparison of cyclegan and unit, 2018.

[98] Yingda Xia, Yi Zhang, Fengze Liu, Wei Shen, and Alan Yuille. Synthesize then compare: Detecting failures and anomalies for semantic segmentation, 2020.

[99] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. 2020. https://d2l.ai.

[100] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. Deeproad: Gan-based metamorphic autonomous driving system testing, 2018.

[101] Qi Zhang, Tao Du, and Changzheng Tian. Self-driving scale car trained by deep reinforcement learning, 2019.

[102] Husheng Zhou, Wei Li, Yuankun Zhu, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. Deepbillboard: Systematic physical-world testing of autonomous driving systems, 2018.

[103] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.

[104] Leon Žlajpah. Simulation in robotics. *Mathematics and Computers in Simulation*, 79 (4):879–897, 2008. ISSN 0378-4754. doi: https://doi.org/10.1016/j.matcom.2008.02.017. URL https://www.sciencedirect.com/science/article/pii/S0378475408001183. 5th Vienna International Conference on Mathematical Modelling/Workshop on Scientific Computing in Electronic Engineering of the 2006 International Conference on Computational Science/Structural Dynamical Systems: Computational Aspects.