CHALMERS
UNIVERSITY OF TECHNOLOGY

Imitation learning for vision-based lane keeping assistance

Master's thesis in Systems, Control and Mechatronics

HENRIK LINDÉN CHRISTOPHER INNOCENTI

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

Master's thesis 2017:023

Imitation learning for vision-based lane keeping assistance

HENRIK LINDÉN, CHRISTOPHER INNOCENTI

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Signals Processing and Biomedical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2017

Imitation learning for vision-based lane keeping assistance
HENRIK LINDÉN, CHRISTOPHER INNOCENTI

Supervisor: Nasser Mohammadiha, Volvo Cars
Lennart Svensson, Department of Electrical Engineering

Examiner: Lennart Svensson, Department of Electrical Engineering

Master's Thesis 2017:023
Department of Electrical Engineering
Division of Signals Processing and Biomedical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Image from the IPG CarMaker software.

Typeset in LaTeX
Gothenburg, Sweden 2017

Imitation learning for vision-based lane keeping assistance
HENRIK LINDÉN, CHRISTOPHER INNOCENTI
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Autonomous driving is currently an active area within the automotive industry where many different solutions are being proposed, for example data driven methods such as deep learning. The aim of this work is to investigate whether neural networks trained by means of imitation learning are capable of acting as an end-to-end solution for lane keeping assistance, using a single grayscale front view camera. The capability and robustness of such models, when exposed to both regular and unexpected driving conditions, have therefore been evaluated. Furthermore, the concept of domain changes, whether knowledge learned from real world driving scenarios is transferable to simulation environments, is investigated.

The model proposed in this work is based on a state–of–the–art neural network architecture. The data from which it has learned stems from human driving collected on highway-like roads under various conditions, such as time of day and weather. The model has been evaluated in simulation environments with realistic road geometries. The results from this work clearly shows that a neural network trained on real driving data generalizes the concept of driving across different domains, performing well in simulated environments. Additionally, at least for simulated environments, empirical analysis on the model robustness seems to contradict earlier results, which suggests that training a model from a single front view camera is insufficient in order to achieve lane keeping functionality. Based on the findings, the conclusion is drawn that neural networks indeed exhibits some capability to work as an end–to–end solution to specific driving scenarios. However, more work, most notably real vehicle tests, are required in order to assert this completely.

Keywords: Autonomous Driving, Deep Learning, Neural Networks, Machine Learning, Artificial Intelligence, Image Processing, Decision Making.

Acknowledgements

We would like to thank the teams at Volvo Cars Active safety for their help with the necessary vehicle specific information required to create the dataset and also the drivers themselves who did the actual logging. Additionally, we would like to thank IPG Automotive for providing and supporting us with CarMaker. Lennart Svensson, our supervisor at Chalmers also deserves a mentioning for his helpful insights during our meetings. Further, our supervisor at Volvo Cars, Nasser Mohammadiha and also Ghazaleh Panahandeh deserve a special thanks for their support in discussing the various angles and aspects of of this work.

Last but not least, to colleagues, in particular Dónal Scanlan, Lucía Diego Solana and Edvin Listo Zec, and family for their moral support throughout the semester.

Henrik Lindén, Christopher Innocenti, Gothenburg, June 2017

Contents

1
Introduction

The human fascination with autonomous machinery has been around for a long time. However, one of the most active areas for machine learning at present time is within the automotive industry, where most major companies are dedicating huge effort into creating vehicles capable of autonomous action, with no human driver required. However, the field is just in its infancy and currently it is not known exactly how to solve the task of autonomous driving, or if it is even possible.

This introductory chapter provides a short overview of the background to the state-of-the-art methods used for autonomous driving. Further, some of the relevant work that has been made within this field by adopting a deep learning approach are presented. The purpose of this thesis is then given, coupled with the imposed limitations required for making the scope of the project feasible. Further, an explicit account for the contributions made in this work are presented, followed by a discussion on ethical and sustainability aspects connected to the project. Finally, this chapter ends with a disposition of the thesis, providing some insight into how the contents of the project have been structured including some guidance on how the thesis could be read.

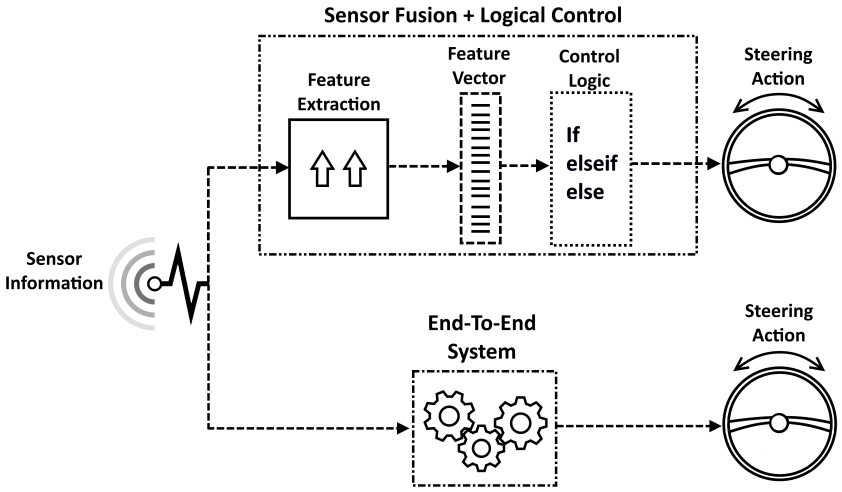


Figure 1.1: A schematic demonstration of the difference between conventional systems for autonomous driving and end–to–end systems.

1.1 Background

In the active safety departments of many car manufacturers, work on sensors and systems for autonomous driving (AD) is carried out. In these applications, usually the raw data from sensors such as camera, radar, and lidar is processed and fused in order to provide an accurate description of the environment in which the car is driven. The output of this processing is usually a list of objects with information describing their status in the environment, such as their position and speed. This information is then utilized to design functions such as path planning, adaptive speed control, or collision avoidance systems. That is, until recently, a potential autonomous driving agent would consist of some large system of logic that makes use of the acquired data in some engineered way.

Manual engineering allows for great flexibility in the design of a system, but it is quite unlikely that it results in an optimal one. Furthermore, to manually engineer relevant features from the raw data is a difficult and time-consuming process. Of course, this also means that it is quite expensive as well. In addition, creating good features requires top knowledge within the respective field which may be difficult to acquire. A way to remedy this problem is to employ a deep learning (DL) approach as it is data-driven with the ability to learn relevant features from the data. Because deep learning models can operate on raw input data, they completely remove the feature engineering requirement and might have the potential to work in an end–to–end fashion. Figure 1.1 illustrates the difference in operation of the two types of systems, where the traditional ones have several stages that must be properly designed as opposed to the end–to–end variants.

The name deep learning stems from the fact that such models are defined by computational graphs comprised of several processing units, stacked in layers. The layers are connected in a long, or deep, structure, allowing the computational model to learn from data at multiple levels of abstraction [1]. Deep learning, in the form of convolutional neural nets, has in recent years provided significant breakthroughs in

the fields of image and video processing, object recognition and speech recognition [2, 3]. However, one drawback of such models is that they require a substantial amount data to learn from, before they become viable.

1.2 Related Work

As the task of making a car fully autonomous is very complex, and because it has been a sought goal for a long time, there have been a lot of efforts made in order to solve the problem. Most efforts can however be divided into two categories that are either based on large control logics (as mentioned in the previous section) or are data driven solutions. Successful examples of systems comprised of modules of control logic are among others, the 2007 *DARPA* Urban Challenge winner: Tartan Boss [4], and the Daimler team that made a Mercedes S 500 drive autonomously [5].

A first step towards a data driven solution was taken in 1989 in the *ALVINN* project [6]. The system used, by today's standards, a very small fully connected feed forward network to directly map camera input from the road ahead into appropriate steering actions. Although trained mostly on artificial data, the system managed to drive a 400-meter path through a wooded area of the Carnegie Mellon University campus under sunny conditions with a speed of 0.5m/s.

In 2005, a new effort for autonomous obstacle avoidance and steering was performed, the *DAVE* project [7]. In this case, the goal was to create a system that could handle off-road obstacle avoidance and navigation without the need for feature engineering. In contrast to *ALVINN*, the system utilized a more advanced neural network architecture, which included convolutional layers. Additionally, it was trained using real world data from a front mounted stereo camera.

Almost two decades after the initial *ALVINN* project, in 2016, the American technology company Nvidia also developed a deep learning based system for autonomous driving [8]. Their project, *DAVE2*, used an even more advanced convolutional neural net (CNN) than *DAVE* and was trained on a larger dataset. The data consisted of everyday driving on public roads, captured from three front mounted cameras, although at inference time the system operates on images from a single camera. As reported by Nvidia, their vehicle managed to drive autonomously 98% of the time in their trials.

From these works it would seem that the use of neural networks as an end–to–end solution for autonomous driving is a promising lead. Seemingly, recent efforts indicate that CNNs should be particularly suitable for the task. In many cases, CNNs have been shown to perform exceptionally well in detection tasks and predictions that are crucial to make in a traffic environment.

It has been suggested in [8] that the use of multiple cameras, are required in order to obtain a system that can stay continuously well positioned on a road. In fact, for the previously mentioned projects that employed a neural network approach, the systems relied on multiple sensors to interpret the surrounding. For the *ALVINN* project, in addition to using camera images, a laser range scanner was also used. Similarly, *DAVE* made use of stereo cameras and *DAVE2* made use of three cameras.

Requiring more than a single camera seems like a limitation, and it would be interesting to investigate further. Additionally, *DAVE* and *DAVE2* also utilized

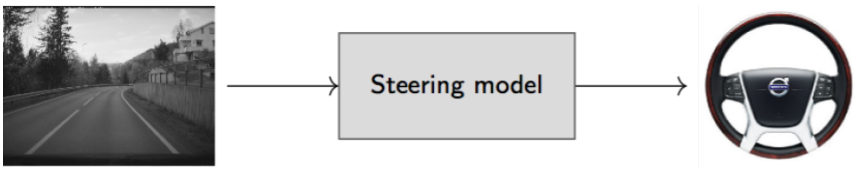


Figure 1.2: Conceptual illustration of the input/output of the intended lane keeping assistance system (the steering model).

color images, which might contain more information than what is actually needed. By instead using a gray scale camera, less information needs to be processed and therefore the complete system might be realizable using lower cost components.

1.3 Purpose

The purpose of this project is to investigate the suitability of neural networks as an end-to-end solution to the process of steering a vehicle in a highway environment. In particular, this thesis will investigate a neural networks capacity to act as a lane keeping assistance system (LKA) using only single grayscale images captured from a front mounted camera for both training and inference. In previously made works, as mentioned in Section 1.2, the systems relied on information from multiple sensors, either during run-time or for training. As images are very rich in information contents, it is interesting to verify if a single front view camera is sufficient for LKA. Additionally, such a system's robustness against sudden disturbances will be investigated as well. Finally, the thesis will consider the models capability of transferring knowledge from one domain (environment) to others. The general idea of how the final system is intended to work is displayed in Figure 1.2. Based on a single image in, the model should produce an appropriate steering command to maintain good positioning in the lane.

1.4 Limitations

As the task of creating a system for autonomous driving comprises much more than what is reasonable to fit in a master's thesis, the scope of the project was limited. Firstly, the target environment for driving was restricted to highway and country road types only. No consideration for city driving was taken, and therefore the speed was fixed to be in the range 50-110 km/h. Additionally, the intended driving behavior was limited to lane following with no other maneuvers such as overtakes or merges considered.

For the learning of the neural network, only (supervised) imitation learning was employed, and the data only consisted of raw images and steering angle measurements without augmentations. The network was limited to control steering only with no other outputs, such as accelerations, considered. For evaluation purposes, simulated environments were used and no real vehicle tests were done.

1.5 Scientific Contributions

In this thesis, the main contributions to the field of deep learning for autonomous driving can be summarized as follows:

1. Demonstrating that, a neural network model trained and operating on imagery from a single front mounted camera is sufficient for lane keeping (in simulated environments), and acquires some robustness. This is to say it can learn to recover from mistakes such as orientation misalignment with the road based only on observing ordinary driving maneuvers.

2. Showing that the neural network model trained on real data is able to transfer learned concepts across different domains, allowing for evaluation in simulated environments. This is an interesting aspect because models showing robustness against domain changes might allow for early development tests to be done in simulation.

3. Providing a basis for discussion whether neural networks may serve as a holistic solution for autonomous driving.

1.6 Ethical Aspects

With most introduction of new technology in society, it affects people in different ways. While the core intent most often is to improve the quality of life for the people using it, there may be some drawbacks worth considering. In this case, the technology in question is a system for autonomous driving. Hopefully, the amount of fatal accidents will decrease as automated systems are introduced because of the fact that they could be much more reactive and attentive than humans. It could also become possible for people to be productive on the go or simply use the time spent commuting for recreation.

However, the introduction would also come with a price. For example, quite a lot of people currently make a living driving various vehicles. If an AD solution proves itself to be as reliable (or more so) than a human driver then this work group may run the risk of getting laid off. The AD system is just a one-time investment and does not require benefits or salaries, making it more preferable than a human employee.

The fear of having new technology coming to replace the jobs of the people has been around at least since the industrial revolution with the introduction of the automatic looms [9]. Since then, many new and fantastic inventions have been introduced, making many old professions completely obsolete. What is important to realize is that with the introduction of new technology many new opportunities for new areas of work appears. Just 50 years ago it would not have been possible to anticipate how many people would be involved within a field such as software development. Yet, today, some of the most successful companies are largely based on this "new" tech that is the modern computer. Maybe the story will be similar with the introduction of autonomous cars.

1.7 Sustainability Aspects

With vehicles becoming autonomous, less resources are expected to be required to accommodate them. With the more exact and reactive automatic driving systems, lanes could be made much more narrow, meaning that more cars fit in the same space. This would allow for more people to travel simultaneously while maintaining a good traffic throughput for the same amount of materials spent on roads.

In connection with this increase in reactivity, perhaps also coupled together with real-time communications with cars in the nearby area, it might also be that the emissions or fuel consumption lowers substantially. This would be the result from the cars simply driving smoother and able to perform better path planning than humans currently can. Furthermore, autonomous vehicles could also very well perform safe "platooning" where many vehicles drive in very close proximity to one another to reduce drag, resulting in even lower fuel consumption. It may also be possible to see a substantial shift in how autonomous cars are used and owned. The concept of carpooling may become more popular where it could become possible for one car to service many users, reducing the total number of cars in the vehicle fleet.

Naturally, none of these good outcomes are for certain. It could very well be that the increase in comfort and ease of using AD vehicles magnifies their popularity. This might increase the demand for such vehicles, resulting in more units which then might consume all the positive effects resulting in a net increase in fuel consumption and emissions.

However, it might be the case that it would not be worse than what is happening at the moment. Today, an increasing number of people are able to afford a car resulting in greater emissions and more frequent and severe traffic congestion's. As autonomous cars become more affordable and convenient, they could also attract the people currently commuting with trains and buses which of course would not be particularly good. Maybe the adverse effects can be dampened with proper regulations on a national or global level, but how this should be done is only speculations at this point.

1.8 Disposition

This chapter has introduced the purpose of this work and provided a background to the area of deep learning in connection to autonomous driving. A few examples of earlier work have been provided, with a focus on projects adopting a neural network approach and potential limitations. The restrictions imposed on this project have further been stated and the main contributions of this work have been summarized. Finally, a discussion on potential ethical and sustainability aspects has been given. The remainder of this thesis provides a more in-depth description of the work that has been done.

– Chapter 2 gives an overview of two simulation environments which have been used for the purpose of model evaluation and data generation. The structuring and processing of data (both the real data provided by Volvo Cars

and the synthetic data obtained from the simulators) is further presented and specifications of the resulting datasets, used for model training, are given.

– Chapter 3 provides the core theoretical concepts of neural networks which are relevant in order to understand the models used in this work. The architecture of the model for end–to-end inference of steering actions is presented and a discussion with respect to its main components is given.

– Chapter 4 describes the theory on how neural networks are able to learn from data and specifies the configuration for the learning procedure. Further, a description on model implementation and model–simulator interfacing is given.

– Chapter 5 deals with the task of model evaluation. The evaluation process is initially explained in connection to the concepts of closed loop simulation and transfer learning. An analysis of the predicted steering actions is presented and the models capability of transferring knowledge across domains is discussed. Then the resulting driving performance, in terms of two proposed performance metrics is given, expanding on the topic of how to evaluate a system which performance largely is subjective.

– Chapter 6 states the conclusions drawn as a result of this work, with the main focus on whether neural networks seem to be suitable as an end-to-end solution for autonomous driving.

– Chapter 7 expands on the subject of model design and provides a discussion on potential model extensions. Many of the ideas presented are based on work that was done in the thesis, but failed to perform adequately due to technical difficulties.

2
Environments and Data

For any data driven model to work in a satisfying fashion, good data has to be available. When such data is unavailable, a large amount of work has to be spent on constructing proper databases before further progress can be made. Whichever is the case, there are different types of data that one can decide to use. One choice is to use real data, such as photos or real measurements. Another option is to use artificially generated data from a model of the physical world such as rendered images from a graphics engine or artificial signals from sensor models. In this project, both types of data were used for model training.

A developed model must often be verified in some way to determine its performance. The verification procedure would ideally be carried out with real experiments but this is unfortunately not always possible. For safety critical systems, virtual evaluation is preferable at early stages of development to reduce costs and potential risks involved with real testing.

In this chapter, the two different simulation environments used for this work will be presented in Sections 2.1 and 2.2. Furthermore, the process of structuring the main dataset from Volvo Cars and making it as vehicle independent as possible is stated in Section 2.3. Finally, the last part of this chapter, Sections 2.4 and 2.5, deals with the process of obtaining artificial data from the simulation environments.

2.1 Unity Game Engine

The Unity game engine [10], developed by Unity Technologies, is a free tool (for educational purposes) for building and to code video games. The engine ships with what is referred to as "Standard assets" that can be used to quickly get a grasp of the engines capabilities. The standard assets consist of various objects and scripts that can be repurposed as the user see fit. Almost anything can be created in the game environment provided that the user has the required time and know-how, making Unity a very general-purpose environment. The Unity community is also quite extensive, with a plethora of additional packages available for download.

Conceptually, the process to create a game starts by defining what is called the "scene". The scene is the actual environment that the game will be played out in. In the scene, various objects can be placed, such as trees, roads, cars, and so on. To make the scene more dynamical, it is possible to attach scripts to most of the objects. Without them, everything would be static and in an unplayable state. Adding the scripts enables things to change based on certain events happening such as the player pressing a particular key. The scripts are coded in the C# or Java Script programming languages. Using scripts, it is also possible to create game applications that interact with other programs. Utilizing this functionality, it is possible to use the Unity game engine for model in the loop (MIL) purposes.

When users have created whatever content they want, the complete package can be compiled into an executable file for many target platforms. For the purposes of testing a self-driving agent for example, one could create a game with a car plus a track and then have the agent try to steer it in real time. While this is a valid approach to take, creating custom games is a very time-consuming process.

While Unity indeed includes some physical properties such as gravity etc., it is still a general-purpose game engine and often utilizes simplified versions of physical interactions and sensors. As such, it is suitable for simple or early MIL testing but if more realism is desired, other environments might be preferable.

2.2 IPG CarMaker

CarMaker, created by IPG Automotive [11] is a real-time simulation software solution developed specifically for virtual testing of passenger cars and light-duty vehicles. The software is capable of modelling the environment of a vehicle with different traffic scenarios, pedestrians, road signs etc. The software also comprises an intelligent driver model, support for detailed vehicle models, and highly flexible road generation.

The simulator uses a highly detailed physics engine, mimicking many of the intricate relationships between physical objects in a realistic fashion. Car models are subject to realistic centripetal forces when turning, friction is based on materials interactions, even sensors can be accurately modeled, being affected by weather or other environmental effects. Finally, as simulations are event and maneuver driven, it allows for both open and closed loop testing.

Driving scenarios in CarMaker can be created by simply designing the road

geometry using segments. By chaining together different segments, such as curves, straight roads and intersections, arbitrarily complicated road geometries may be created. The different segments can then be customized with proper lane markings, traffic signs, road types etc. One can also include a variety of triggered events in the simulation, such as added disturbances in order to test model robustness. Scenarios can also include different types of vehicles, each following specific trajectories at different speeds. Naturally, the vehicle to be controlled is included, where model functionality can be added. For example, a new steering model for autonomous driving.

Road geometries are not restricted to perfectly straight lines and curves. It is possible to base the road geometry on real data from longitude and latitude coordinates. This is done by specifying keyhole markup language (KML) files which can be imported into CarMaker. Although this captures the geometry of the road, objects along it such as houses, trees etc., have to be manually modeled.

An animation of the simulation can play out in real time with the possibility of being used as a base for any vision systems including virtual cameras. The cameras can be adjusted using a wide variety of lenses and configured to record at an arbitrary resolution and sample frequency. The captured images can be transmitted out of the simulator for MIL purposes, or used internally with any included code.

2.3 Volvo Expedition Data

In this project, the main data to be used for training the neural network models was supplied from Volvo Cars. The available data consisted of a large collection of log files from various expeditions performed over the last 5 years. These expeditions usually had the purpose of testing some new functionality for future cars and to collect data that could be used for development.

The recorded information contained for example the video-feed from the vehicle's front mounted camera, the states of the vehicle sensors etc. For the purposes of this project, the extracted portion of the data was limited to the captured video frames and the sensor measurements from the steering wheel, the car's current speed, acceleration and yaw rate. The image data in this case was extracted as 640×480 gray scale. Figure 2.1 shows a selection of images captured from various Volvo expeditions under a variety of road and weather conditions.

The vehicle state sensors had a sampling rate that was roughly three times higher than that of the video feed. This difference in rate posed a small problem, as there could be multiple sensor readings per image frame. To remedy this issue, the sensor measurements that most likely corresponded to particular frames were extracted. This procedure was done by matching the timestamps of the videos and the sensor readings to find the closest match. The readings that did not get paired up with a frame were then simply disregarded.

In order to attain data driven models that generalize properly, ideally a large and very diverse data set to train on is very beneficial, almost required even. This is especially true if one would like the model to be robust to sudden changes in the environment (entering new areas or road types) or being able to react to improbable events. For example, if the model never has observed how to avoid obstacles on the

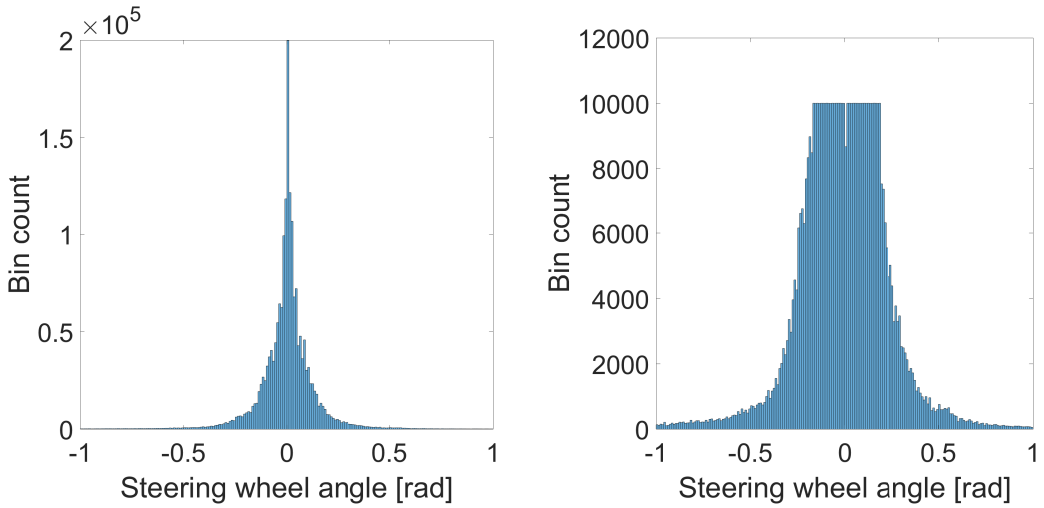
Figure 2.1: A selection of images from various Volvo expeditions under a variety of road and weather conditions. Note that the images here are cropped as described in Section 3.2.1.

road, it will be very unlikely that it would do so in the live setting. Having a large and diverse dataset is one way of ensuring that the network gets to observe as much diverse situations as possible.

As the target environment to drive in was to be country- and highway roads in Scandinavia, some manual filtering of the Volvo data was made. Only data captured from the relevant road types in Sweden and Norway were chosen. Furthermore, some effort was made in order ensure that the amount of city driving that was included in the dataset was minimized. As the dataset available at Volvo cars that met these criteria still was overwhelmingly large, manual inspection of the videos to determine their suitability was out of the question. Therefore, the logs were filtered based on data-viability. Viable logs were to contain no lane changes, and with not too slow traffic (driving below 50km/h) as that was likely to be city-driving. Also, the vehicle was only ever allowed to be driving forwards.

The filtering procedure produced a list of logs that passed the requirements, with information about the mean, standard deviation, and variance of the steering wheel angle (SWA) used throughout each particular log. This allowed for a sorting to be done post filtering. In this first sorting, only videos with a SWA variance of at least 0.0025 radians were picked. This way, roads involving somewhat frequent amounts of turning were favored over the ones that mostly contained straight driving.

The following sections describes how the pruning process of the data was made in order to reduce bias toward any particular range of the SWA. Additionally, an account for how the transformation of SWAs to a more vehicle independent measure was performed is given, and the section ends with a discussion on the resulting dataset.



(a) The complete dataset. **(b)** The pruned dataset.

Figure 2.2: Histograms of steering wheel angles from the complete dataset (left) and the pruned dataset (right). Histograms depict the significant range $[-1, 1]$ rad where most samples lie, rather than the complete range $[-9, 9]$ rad.

2.3.1 Pruning

Having collected the initial dataset, it became apparent that even though curvy roads were favored over straight ones, the data still showed some bias. Seemingly, when driving a vehicle, the actual turning of the wheel is quite limited. Because the Volvo data came from real driving scenarios, it naturally showed the same characteristics as ordinary driving do. That is to say, the case of driving straight was overrepresented. While the aspect of driving straight ahead indeed is an important part of driving in itself, the fact that this was so prevalent posed a problem when using the data to train models.

To reduce the bias, in this case driving straight, one option was of course to filter out some range of values that considered to be within the range of driving straight. While this indeed would have resolved the overrepresentation, it was likely to instead induce the opposite problem. Because then the model would never have learned to drive straight, it might have ended up steering too much and too frequently, never driving smoothly. Naturally, this was not a desirable outcome either. Another way of ensuring that the data had roughly equal representation over the entire domain was to prune it. For this project, this was the preferred solution because collecting more data to reduce the overrepresentation of straight driving was not an option.

To get a feeling for how the collected dataset was distributed, a histogram was created, using 18000 bins, over the full range of the steering wheel angle, [-9, 9] rad. From Figure 2.2 it is clearly visible that the data was centered around zero radians. Furthermore, the data was residing on a quite narrow part of the actual range of possible steering angles. The conclusion was then drawn that training a model on such a dataset would be detrimental to its performance.

In order to remedy this issue, fair pruning of the dataset was performed. Fairness in this regard meant that each expedition was appropriately pruned so large

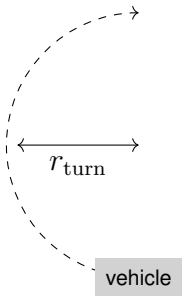


Figure 2.3: A turning circle of a car when driven with a constant steering wheel angle α. The radius r_{turn} of that circle correspond to the vehicles turn radius.

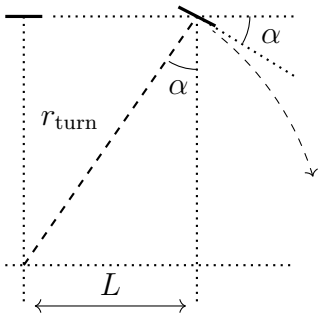


Figure 2.4: A bicycle representation for calculation of the turning radius r_{turn} of a vehicle with wheel base L and front wheel angle α.

expeditions, with many data points, would have more image–steering pairs removed than the small ones. This ensured that as many different settings of the environment was left in the final dataset, improving its diversity. The level of the cut was set by specifying the maximum number of occurrences (image-steer pairs) allowed in any of the 18000 bins in the histogram. The histograms of the complete dataset as well as the pruned dataset can be seen in Figure 2.2.

2.3.2 Transformations

Nearly every car model, or at least every brand, has a different range and a different conversion rate from its steering wheel to the front wheel angles. Therefore, using the raw recording of the SWA as a measure would mean that a model might become dependent on the actual car used to collect the log data (in this case the Volvo XC90) and not work properly on other vehicle models. To circumvent this issue, making the output of the model vehicle-independent, a transformation of the SWA was made. Instead of using the SWA directly, translating it to the turning radius and subsequently to a measure of curvature (the inverse turning radius) was deemed a better alternative which could be used for any vehicle.

The turning radius is the radius of the turning circle a car will make provided a constant angle on its steering wheel as seen in Figure 2.3. Clearly, this radius ranges from infinity to some minimum value that the car is mechanically limited to go below. If the model could output such a value, every car that it was to operate in would just require some vehicle specific transfer function to map the signal back

to a proper SWA.

To transform the SWA to the turn radius, one first has to transform it into the angle of the front wheels and then use a vehicle model for further transformation. In this case, a transfer function from the XC90 SWA to its front wheels was approximated by a third-degree polynomial based on data provided from the Volvo vehicle dynamics department. Having obtained the average wheel angle, the turn radius was calculated by approximating the car with a simple bicycle model. The turn radius could then be represented as shown in Figure 2.4. Using simple trigonometric formulas, the turn radius was deduced as

$$r_{\text{turn}} = \frac{L}{\sin(\alpha)} \implies \frac{1}{r_{\text{turn}}} = \frac{\sin(\alpha)}{L} \tag{2.1}$$

where α is the front wheel angle, L the wheelbase, and r_{turn} the turn radius.

Because the turn radius ranges from some minimal value to infinity, it might in some cases be problematic to represent it properly in a computer. Therefore, using its inverse might be more beneficial. For the XC90, this transformed the range of possible values to approximately $[-0.17, 0.17]$ m^{-1}, easily represented digitally in a computer.

2.3.3 Resulting Dataset

The previous sections described the pruning and transformations made to the raw dataset in order to improve the quality and usability of the models. As the training data was based on camera images, it could not be made entirely independent of the host vehicles' camera placement.

When running the model in simulations or in reality, this meant that the camera had to be aligned with the camera positioning that was used when the images in the dataset were recorded. Otherwise the model's different perspective of the world would most likely cause it to make faulty predictions. This is because with the new perspective, objects of interest were no longer as likely to appear in the way the model had learnt to recognize them. Having a dataset that included many different examples of camera-mounting would most likely have enabled the model to generalize beyond such a problem, but in this case, as the data was already recorded, such changes could not be included.

For each of the chosen videos from the first viability filtering, as described in Section 2.3, the video was split into its separate frames and paired with the corresponding sensor reading (SWA). In the end, this netted a total dataset of 1.4 million image-measurement pairs. Based on the sampling rate, this amounted to approximately 27 hours of driving.

2.4 Unity Data

For verification purposes, training a model solely on Unity data was performed in order to ensure that the model itself was sound. Realizing that manually driving the car in the Unity game engine would be a very time-consuming process when collecting a large dataset, a method for automatic data collection was created. In

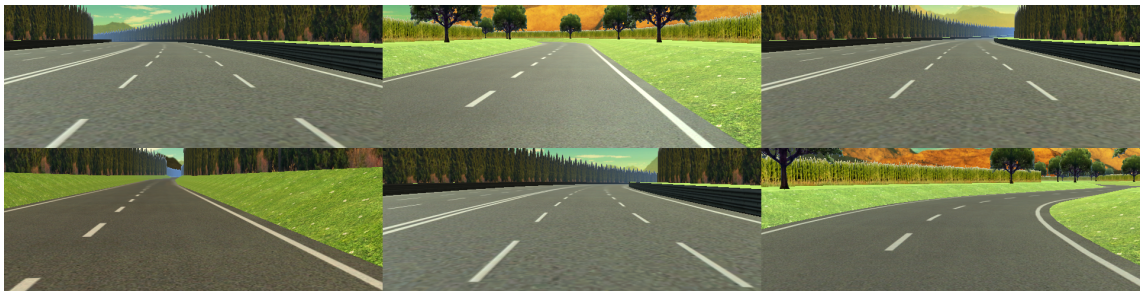


Figure 2.5: A selection of images collected from the Unity game engine environment illustrating a variety of lane configurations and backgrounds.

order to make the car drive automatically, parts of the standard assets were used. A set of 400 waypoints were manually put along the route that the car should drive. The collection of waypoints together defined a line that the car could be set to follow. By varying the parameters, such as how much the car was allowed to cut corners, changing the initial position of the vehicle, and its maximum speed, different diving behaviors could be attained. This meant that it was possible to ensure some diversity in the dataset. In total, roughly 50 000 image-steering pairs, or about 45 minutes of driving, were collected. A selection of images from the dataset can be seen in Figure 2.5.

As this dataset was quite limited in terms of its size compared to the one extracted from the Volvo logs, it was not put under the same pruning regime as in the other case. Doing so would have meant that most of the data had been pruned away, which was not desirable. The recorded steering angle was not transformed as before, because the model to be trained on Unity data was only intended to be used in Unity were the recorded SWA could be used directly.

2.5 CarMaker Data

For the same reasons as with the Unity environment, a model trained on only CarMaker data was to be created for verification purposes. For this case, data from a real-world road geometry was collected. The road geometry was a stretch of roughly 53km country road from the southwestern part of Sweden, Riksväg 160 from Rotviksbro to Ucklum. The choice of this particular road was based on the fact that it provided a good mix of different scenarios. Long straight sections and wide curves mixed with winding sections and moderately sharp turns. Also, there were two roundabouts in the original road which was edited out as they were deemed to fall outside the scope of the highway and country road type.

With the road geometry specified, the included IPG driver agent was configured to traverse the complete road, staying roughly in the middle of the lane at all times. The IPG driver agent is essentially a model of a driver which can be tuned to behave realistically depending on the tuning of its parameters. The parameters of the agent were appropriately adjusted to ensure realistic cutting of corners and lane keeping with the speed of the driver set to a constant value of 70 km/h. The front mounted camera in the vehicle was adjusted to conform with the specifications from the Volvo

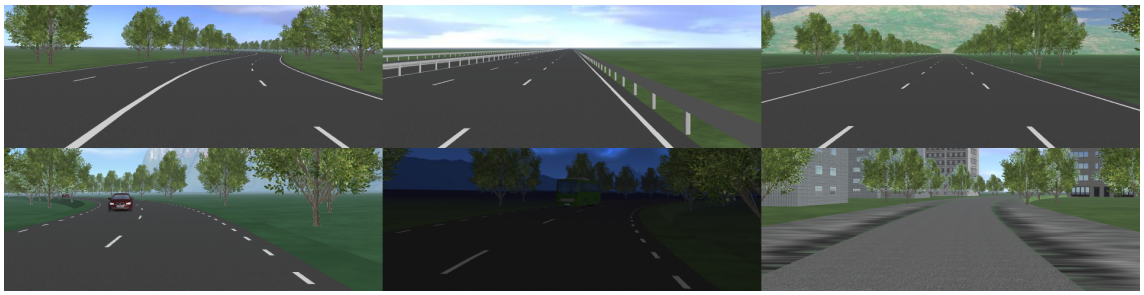


Figure 2.6: A selection of images collected from the CarMaker environment illustrating a variety of lane configurations, lighting and background scenarios.

log, both in terms of physical positioning but also with regards to resolution and sampling rate.

With all the specifications made, data was logged in a similar way as it was done in the Unity engine. In total, the data from CarMaker amounted to approximately 50 000 image-measurement pairs, equivalent of roughly 45 minutes of driving. A selection of images can be seen in Figure 2.6 and as with the Unity data, this dataset was deemed too small to undergo the pruning procedure.

3
Model Design

The human brain is a curious thing. Based on the intricate network structure between its many neurons, it is capable of learning to perform very complex control tasks with ease. It is for example able to maintain balance on a bike dashing at full speed through a forest, or navigating a vehicle in a highly complex urban environment using almost only vision data as input. Creating an autonomous agent for such complex control tasks has been a sought goal for quite a while, with varying degrees of success so far.

Defining some sort of control logic, by manually writing a set of rules that enables such complex behavior is not easy. The brain does not seem to possess these capabilities inherently but apparently learns them by observing data from the world around it through the many sensors that the body contains. If an autonomous agent too could learn by experience and example, the whole design of the control logic and how to make use of the sensor information is alleviated. In fact, some of the current state–of–the–art systems for autonomous driving are actually data-driven and have learned the task of driving based on data they observe.

In this chapter, an overview of the design of the neural network models used in this project is provided. In Section 3.1, a quick introduction, coupled with some history, is given. Additionally, the core components that make up neural networks are displayed and their workings explained. Then, the section provides some detail on the most basic neural network models, moving up to the general idea of CNNs. The chapter ends with Section 3.2, where the proposed model architecture for the task at hand is given.

3.1 Artificial Neural Networks

Neural networks are one of the several methods that has been tried in order to attain intelligent systems, or AI. The concept of neural networks was introduced during the 1940s when Warren McCulloch and Walter Pitts created the first neural network model [12]. Initially, the networks were seen to be an analogue to the human brain as they have similar structure and behavior. However in recent years, their resemblance to their biological counterpart has been debated.

At the lowest level, a neural network consists of independent neurons. For biological neurons, each unit has several input gates, in the form of *dendrites*, which feed information into the cell body. From the cell body extends a long connection called an *axon*. This axon in turn connects to other neurons' dendrites, forming a network structure. For a schematic representation of a biological neuron, refer to Figure 3.1.

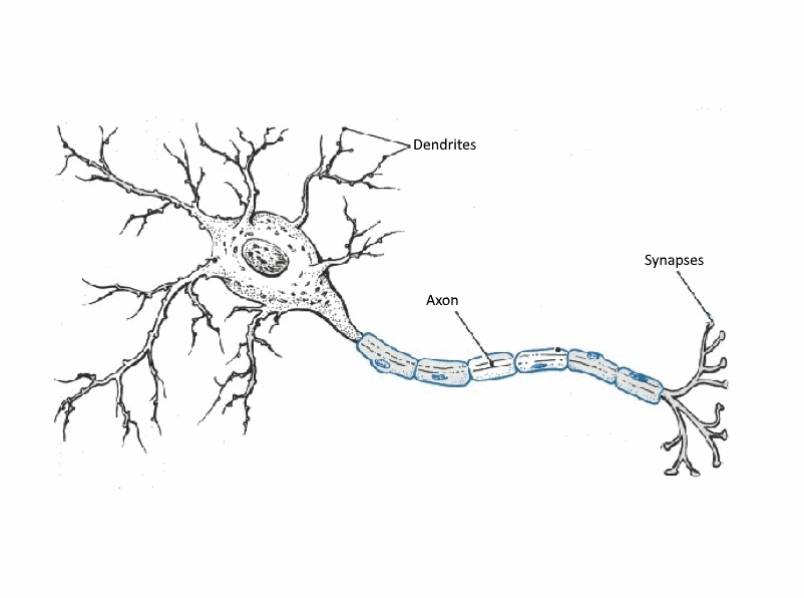


Figure 3.1: An image of a biological neuron [13].

Depending on its inputs, a neuron may pass an electrochemical impulse along its axon. This is often referred to as the neuron being *activated*. The transmitted impulse may in turn activate other neurons that then transmits their signal and so on. Some of the inputs to the neuron may excite it to transmit its signal and other may inhibit it. In addition, some connections seem to play a more central role than others whether the neuron activate or not, seemingly having a larger weight.

The artificial analogue behaves in a similar fashion, albeit a bit more simplified. The central part essentially acts as a small computational unit. It sums up all its inputs x, multiplied by the weights w on the connections, adding a bias term b and finally applies a so-called activation function on the result, outputting some value. That output is then given to all other neurons it is connected to. In Figure 3.2 a single artificial neuron is shown with all of its components displayed.

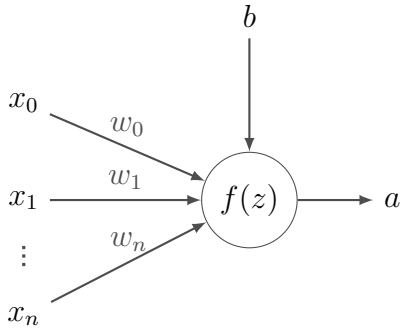


Figure 3.2: An artificial neuron with input x_i, weights w_i, bias b and output a. $f(\cdot)$ is the activation function and z corresponds to the weighted sum of the inputs as described in Equation 3.1.

The output a of a neuron can be written as

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b = Wx + b \tag{3.1}$$

$$a = f(z) \tag{3.2}$$

where n is the number of inputs to the neuron and $f(\cdot)$ is the activation function. The input of a neuron is generally represented in vector form, x, and the weights as a vector W, simplifying the notation slightly.

The remainder of this section will further describe common activation functions, simple neural network models and finally more complex architectures in the form of convolutional neural networks.

3.1.1 Activation Functions

The number of different choices that can be made for an activation function in a neuron is in theory neigh limitless. However, because the neurons usually operate on real numbers, the activation function has to be defined such that it takes a real value as input. Despite the broad possibility of functions to choose, there are some that have been proven over the years to provide good results in different contexts. As of now, the most common ones are the identity, hyperbolic tangent, sigmoid, rectified linear, and the exponential linear functions. See Equations 3.3 for the mathematical representation and Figure 3.3 for plots of the different functions and their derivatives.

$$
\begin{aligned}
\text{Identity:} \quad & f(x) = x \\
\text{Hyperbolic tangent:} \quad & f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \\
\text{Sigmoid:} \quad & f(x) = \frac{1}{1 + e^{-x}} \\
\text{Rectified Linear:} \quad & f(x) = \max(0, x) \\
\text{Exponential Linear:} \quad & f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases}
\end{aligned} \tag{3.3}
$$

In the early days of neural networks, the tanh and sigmoid functions were commonly used. In contrast to the identity function, both the sigmoid and tanh saturate on most of their defined domain and thus only have sensitivity to the input when it is close to 0. In addition, the derivative of those functions also approaches zero for much of their domain, making neurons using them a bit difficult to train using the common gradient methods.

The last two activation functions, the ReLU and the ELU, are nowadays likely the most common ones in the standard, feed-forward, form of the neural networks. The ReLU is composed of two linear segments making it piece-wise linear. The ELU is similar but with the difference of having a scaled exponential function for the negative domain.

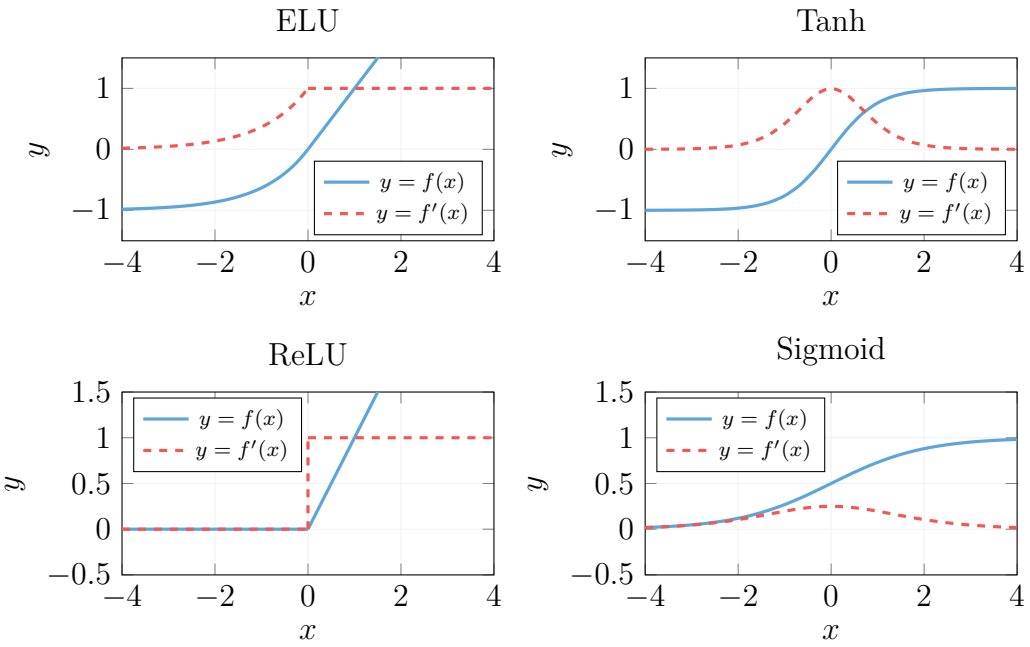


Figure 3.3: Common activation functions and their derivatives including the exponential linear unit (ELU), rectified linear unit (ReLU), hyperbolic tangent (Tanh) and the logistic function (Sigmoid).

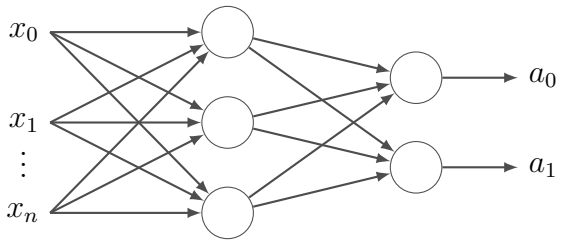


Figure 3.4: A fully connected feed forward neural network with one hidden layer consisting of three hidden units, the input x_i, and outputs a_0, a_1.

3.1.2 Fully Connected Feed Forward Neural Networks

Fully connected feed forward neural networks (FCFFNN) were in their initially simple form developed in 1957 by Frank Rosenblatt for the US Navy [14]. However, at the time, the networks were actually modeled as a physical machine rather than a software. The FCFFNN structure is likely the most common topology a neural network can have. Here, the neurons are structured in layers to form a directed acyclic graph.

Every neuron in each layer only feeds its output into the neurons in the layer directly after. Similarly, the neurons in a layer only gets their inputs from the neurons directly before it. Furthermore, there are no connections between neurons in a single layer. Only the forward connection is present. Common for all feed forward networks is that the only interface they have with the outside world is the input to the very first layer and the outputs from the very last one. The remaining neurons, between the input and output layers have no means of outputting any value or receiving any value from the outside. Due to this property, those layers are often said to be hidden [15]. In Figure 3.4, a FCFFNN with a three-unit hidden layer is shown.

As opposed to the single neuron, the weights W for a given layer are no longer just a single vector but instead a matrix. Calculating the output of the FCFFNN is referred to as the forward pass, due to the fact that the output values from all of the neurons flows forwards through the network. Assuming the input to some layer l of neurons being $a^{(l-1)}$, the weight matrix for that layer being $W^{(l)}$, and the activation function to be $f(\cdot)$, one can write the output from the layer as

$$z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)} \tag{3.4}$$

$$a^{(l)} = f(z^{(l)}) \tag{3.5}$$

where the output a of the layer now generally is a vector. This vector is then fed into the next layer following the same principle. This procedure is continued until one reaches the final neuron(s).

3.1.3 Convolutional Neural Networks

The convolutional neural network, often shortened to just convolutional network, or simply CNN, is a specialized form of the standard feed forward neural network. It

is tailored to operate on data that has a known grid-based topology such as that of images or audio recordings. In recent years, the convolutional networks have been hugely successful in a quite diverse set of practical applications. Areas in which they have been very successful include, for example, computer vision and voice recognition.

Real implementations of convolutional nets make use of the mathematical operation of convolution, which is from where their name originates. The use of the convolution allows them to be efficient despite the usually quite large data they operate on. This is because the weights in a CNN are shared, rather than each neuron having its own set of weights.

The convolutional operation is defined to be valid between two functions that take real-valued arguments and is also commonly referred to as filtering. A typical example of a convolution is the application of a smoothing effect on recorded noisy signals [15].Assuming two time continuous signals $x(t)$ and $W(t)$, the convolution operation is defined as

$$z(t) = \int_{-\infty}^{\infty} x(a)w(t-a)da. \tag{3.6}$$

The operation is most often shortened using an asterisk as

$$z(t) = (x * W)(t). \tag{3.7}$$

In this setting, one says that $x(t)$ acts as the *input* and $W(t)$ as the *kernel*. There also exists an equivalent version for discretized signals, similarly defined as

$$z[t_d] = \sum_{a=-\infty}^{\infty} x[a]W[t_d - a] \tag{3.8}$$

where t_d is the discretized version of t.

In most machine learning (ML) applications, one assumes that the signals to be processed have a finite duration. That is, outside of a certain time span the signals are considered to be constantly zero. This makes the infinite sum in Equation 3.8 computationally tractable. Furthermore, in many applications, the convolution is performed over several dimensions at once, for example when convolving 2-dimensional data such as imagery etc. The 2D discrete version of the convolution is similarly defined as

$$z[n, m] = \sum_{a=-\infty}^{\infty} \sum_{b=-\infty}^{\infty} x[n, m]W[n - a, m - b]. \tag{3.9}$$

This method of performing the convolution becomes important in the many applications of digital image processing and can naturally be extended to also run on higher dimensions such as for RGB color images which in fact are three dimensional structures.

A CNN takes a tensor (multi-dimensional array) as input. For computer vision tasks, this most often corresponds to a 2D (grayscale) or 3D (color) image. The input is then convolved with at least one *kernel*. The kernel is simply a tensor, where all the weights are stored. For 2D convolution, the kernels are spatially limited in terms

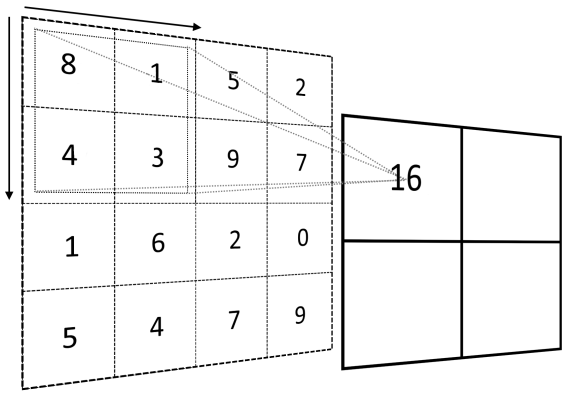


Figure 3.5: A schematic view of the convolution process for an input volume. In this case, a *Full* convolution is performed with a 2×2 kernel with weights equal to one, resulting in a 4×4 output map.

of their width and height, but they always extend throughout the entirety of the input depth. The convolution between the input and the kernel can conceptually be thought of as having the kernel slide over the input volume, calculating a weighted sum of the pixel (input) values.

The simplest setting of the sliding process is when the kernel uses a step of one, referred to as the convolution using a *stride* of one. This means that to generate the output, the kernel moves one pixel to the side after each of its applications. When it has reached the end, it moves just one pixel down and then begins anew until the complete input has been processed. Any integer number is usable for the stride however, and with larger strides the output of the convolution will become smaller. This can actually be used as an efficient down sampling operation as setting the stride larger than one gives the same result as a traditional down sampling of the output from a stride-1 operation.

All of the values produced by the sliding kernel are passed through an activation function similarly as for the FCFFNN. This results in a new 2D matrix of values, a *feature map*. For every kernel that is used, one of these feature maps is produced. As the final step, they are all concatenated, or stacked, into a so called *feature volume*. This feature volume can then be used in subsequent steps, that is, it can be convolved with other kernels to produce new feature volumes which in turn can be used by yet other kernels and so forth. Each section which takes an input and outputs a feature volume is called a *convolutional layer*. Each layer may have an arbitrary number of kernels associated with it, provided they are of the adequate size to handle the input. Figure 3.5 shows the convolution procedure.

More operations can be applied to the feature maps and/or volume in order to attain various effects. One very popular choice is the use of *pooling*. Pooling means that one only extracts a subset of the values from the feature map to be used for the next set of convolutions. This subset is usually obtained in such a fashion as to portray a summary statistic of the original feature map. Oftentimes the pooling procedure can make the network invariant to small translations of its input, meaning

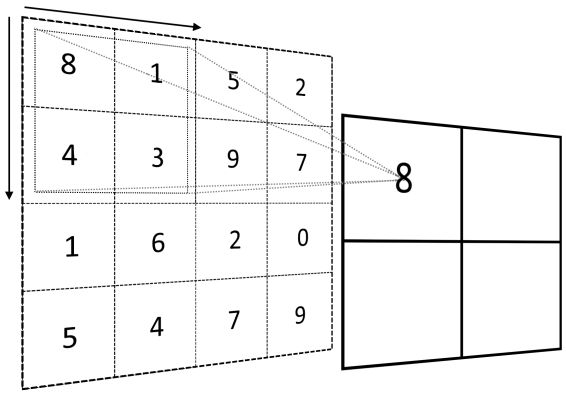


Figure 3.6: A demonstration of the max pooling procedure.

that it reacts to *if* certain features are present rather than where in the input they are located. Figure 3.6 shows an example of a common pooling procedure, the so called max pooling.

Finally, the process of finite convolution may in some cases pose a bit of a problem. If the whole kernel ($k_h \times k_w$) is to fit inside the image when performing the convolution operation, using a stride of 1, the output feature map will shrink with ($k_h - 1$) pixels in height and ($k_w - 1$) pixels in width for each convolutional layer. This means that either the spatial extent of the image shrinks rapidly, or one is required to use very small kernels at each layer. Both of these actions severely limit the capabilities of the net [15]. Fortunately, this issue can be remedied by the use of *padding*. Padding simply means that additional data is appended to places where the kernel otherwise would not have fit, such as in the boundaries of an image. There are varying choices of what to pad with, but the most common choice is often to just add zeroes, so called zero-padding. The extent of how padding can be done ranges from three common configurations, *valid*, *same*, and *full* convolution.

In the case of the *valid* convolution, no zero padding is used at all, meaning that the entirety of the kernel has to fit in the feature volume. In turn, this means that the output will shrink at every layer. The second choice, the *same* version, means that just enough zeroes are added to the data so that the output of layer gets the same size as its input. This allows for an arbitrary number of convolutional layers, as the previous ones does not impact the premises for the next in terms of available pixels. However, the values at the borders influences less output values and thus become a bit underrepresented. The last choice is the *full* convolution. Here, the generated output from the layers will have its width increased by $k - 1$ values as enough zeroes are added such that every value can be visited k times by the kernel. This usually means that the border values of the output are generated by fewer input values than the ones in the center, making it troublesome to attain a kernel that performs well everywhere.

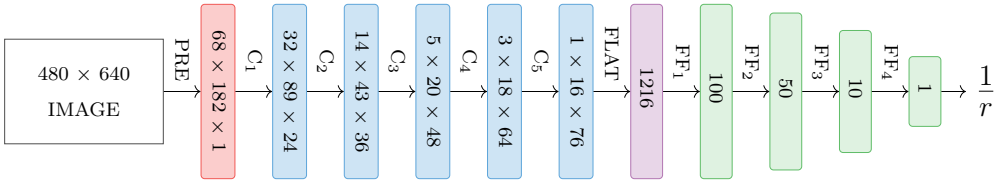


Figure 3.7: Illustration of the baseline steering model architecture. Boxes corresponds to feature volumes after each processing layer, specifying its size. Edges between boxes denotes the applied operations: PRE for preprocessing, C_i for convolutional layers, FLAT for reshaping and FF_i for fully connected layers.

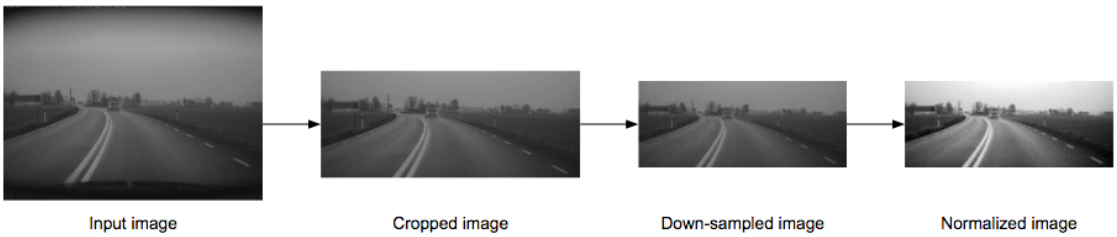


Figure 3.8: Illustration of the preprocessing stages of the baseline steering model. An input image is cropped on top and bottom, down sampled and normalized.

3.2 Steering Model

The model developed in this project is a CNN with a structure similar to the model presented in [8], with the main differences being the data, preprocessing, regularization and training procedure. Minor differences in kernel and hidden unit configurations are also present. The model takes a front view image as input and consists of three main parts. These include a static pre-processing layer, a section of five convolutional layers and lastly a section of four fully connected layers where the output is a single unit (neuron) whose output represents the desired steering action. In this case, the desired steering action is the inverse turn radius which should result in the vehicle staying well positioned on the road. The model structure is illustrated in Figure 3.7 and the individual parts are described throughout the remainder of this section.

3.2.1 Preprocessing

The preprocessing component, although static and not updated during the learning phase, was included in the network structure in order to leverage fast (GPU based) processing routines. Preprocessing was included partly to improve the structure of the input data, enhancing the information, but also to convert the input to an appropriate representation for the remainder of the network. The operations applied in the preprocessing layer can be seen in Figure 3.8 which includes cropping, down sampling, type casting and normalization.

Cropping removes parts from the top and bottom regions of the input image, effectively reducing the computational load. In addition, the amount of redundant information present in the image will also decrease. 35 percent of the top of the

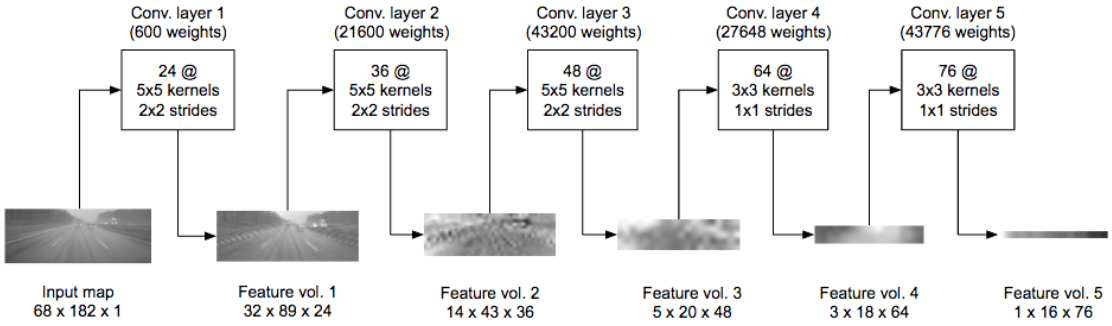


Figure 3.9: Illustration of the convolutional layers of the steering model. The size of each feature volume is specified below an illustration of the corresponding averaged feature volume. The number of kernels, kernel size and stride are specified for each convolutional layer.

image is removed because this part almost exclusively depicts sky and was deemed redundant for the task of determining steering actions. Likewise, 15 percent of the bottom of the image is removed based on the same arguments as this region exclusively depicts the hood of the vehicle. In total, the cropping operation thus reduces the size of the input by 50 percent. The reason for not cropping even more of the height is that it should be possible to see the road ahead even when the vehicle is driving in for example hilly terrain.

Further, the size of the model and the amount of computations needed is reduced by down-sampling the cropped image with a factor ~ 3.5 using bilinear interpolation. This results in an image with a spatial dimension of 68×182. The down-sampled image is then subject to type casting because different sources of input data produces images with varying numerical precision. The different datasets used during the optimization process of this model had image precisions with integer representations that varied from 1 to 4 bytes per pixel for both the 2D (grayscale) or 3D (color) images. As such, the integer representation of the images was converted to 4-byte floating point precision for consistency and to conform with the required precision for the remainder of the computational layers.

Finally, a per image normalization is performed by element wise subtracting the mean and dividing by the standard deviation of the pixel values for each processed image. This procedure was intended to even out the difference between images captured at different times of day, such as night and noon.

3.2.2 Convolutional Layers

The preprocessed image, or *input map*, is passed to the convolutional part of the network, consisting of five layers. Figure 3.9 provides a specification of the convolutional layers and illustrates the average *feature volumes* as outputted by each layer. The process performed at each layer is a strided 2D convolution, followed by ELU activations resulting in 3-dimensional feature volumes.

The kernels associated with the three first layers were set to size 5×5 in the spatial dimensions while spanning the entirety of the preceding feature volumes in the depth dimension. The kernels of the following two layers were similarly defined

with a size of 3×3. Kernel sizes were partly selected based on empirical tests where a 5×5 window seemed to provide a good trade-off between perceptive field, computational complexity and model size in the shallow layers of the network. In addition, this has been shown to work well in earlier work [8]. The reduction in kernel size towards the deeper part of the network was introduced to keep the model size small in addition to keeping the perceptive field from becoming too large. The reasoning behind this choice was because information at the input space effectively gets compressed for each consecutive layer, allowing a 3×3 kernel to "see" more information at a deeper level than a 5×5 kernel would see at a shallower level.

Strides were similarly selected based on empirical evidence. Layer one to three applies 2×2 strided convolution which effectively shrinks the output feature volume to roughly half the size of the input volume for each consecutive layer. The result is thus that the number of convolution operations needed is halved for each layer. Layer four and five uses a 1×1 stride since at this point the feature volumes are small enough and do not require further down sampling in order to have fast inference time.

To keep the expressiveness of the model, the capacity for each layer was extended by an increase of the number of kernels as the feature volumes shrinks in the spatial dimension. In the first layer 24 kernels were used. Then for each of the following layers, additional kernels were added. This resulted in layer 2 having 36 kernels, layer 3, 48 kernels, layer 4, 64 kernels and layer 5 having 76 kernels. The choice of increasing the number of kernels in the fifth layer more compared to the increase done in PilotNet stems from the desire to retain as much feature information as possible in the net.

3.2.3 Fully Connected Layers

The fully connected (FC) part of the model was designed to map the relevant features obtained in the last convolutional feature volume to the appropriate steering action. The process firstly consists of reshaping the feature volume from the convolutional part into a vector representation. The feature vector is then successively mapped to lower dimensional spaces where each consecutive layer represents features of a higher abstraction level. Figure 3.10 provides a specification of the FC layers. In layers 1, 2, and 3, ELU activations are applied to their respective outputs while in the fourth layer, the identity function is used as activation. This last choice of activation function was based on the desire in order to have sensitivity over the complete range of the output.

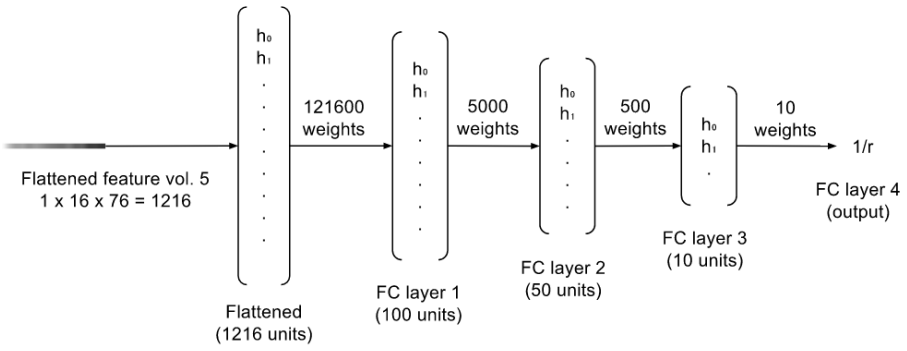


Figure 3.10: Illustration of the fully connected layers of the steering model. The output of the last conv. layer is flattened before being fed into the first layer. The feature is gradually reduced in size as its abstraction level increases, until it finally is summarized as the $\frac{1}{r}$ value.

4

Learning and Implementations

As the human brain gradually learns to perform a multitude of tasks, over many years of experience and observations, so too must an artificial neural network learn to perform its intended task. The process of learning for an artificial neural network consists of gradually making changes in the weights of its connections in order to more closely resemble some intended function.

In this chapter, the procedure for this learning process is presented in Section 4.1 where some of the methods for improving the efficiency of the training are mentioned as well. Section 4.2 demonstrates the way in which the learning procedure has been implemented and how the data was structured. Finally, in Section 4.2.4, an overview of how the trained model was integrated into the aforementioned simulation environments is given.

4.1 Network Optimization and Regularization

Learning in a deep learning (DL) context refers to optimization of some performance metric P, which is often implicitly optimized by minimizing a cost function J. In a supervised setting, a typical approach is to define a cost function over an average of training data x and corresponding target values y as

$$J(f(x,\theta),y) = E_{(x,y)}[L(f(x,\theta),y)] \tag{4.1}$$

where training and target data stems from an empirical data generating distribution. Here, L denotes a cost function w.r.t. a single data point x and $f(x,\theta)$ denotes the inferred output of a deep neural network (DNN) parametrized by θ [15]. In an optimal setting, the objective would be to minimize a cost function where the expectation is taken w.r.t. the true data generating distribution rather than the empirical distribution obtained using a training dataset [15]. In the context of this work, the empirical distribution differs from the true distribution since the training data does not contain all possible driving scenarios.

The most actively used optimization methods for training DNNs consists of variations of the gradient descent algorithm (GD) [15]. GD is extensively used as is, but also variations of it including for example the method of momentum [16] and more recent methods which apply adaption of the learning rate during run time. Adaptive methods include among others the AdaDelta [17] and Adam [18] algorithms. However, the conceptual idea of the gradient descent algorithm is illustrated in Figure 4.1, where one can observe how the minimum of the surface is iteratively found by taking small steps in the negative direction of the gradient.

To improve the rate of which a network learns, the method of regularization is often applied. This is a general method that applies to most ML topics and the idea behind it can be expressed as *"any modification made to a learning algorithm that intends to reduce its generalization error but not necessarily its training error"* [15].

For the remainder of this section, a few concrete examples of cost functions that are commonly used will be presented, moving over to an overview of the most common methods used for optimizing such functions. A short explanation to the approach of supervised learning is then provided and finally, the method used for regularization will be described.

4.1.1 Cost Functions

Deep neural networks can generally be applied for both classification as well as regression tasks. That is, if the desired output should be interpreted as a probability distribution over hypotheses (classes) or represent real valued quantities, DNNs can serve as powerful tools. A cost function which is commonly used for classification problems is the cross entropy. The cross entropy indicates the distance between what the DNN predicts and the ground truth (target) distribution. The cross entropy for a single example x is defined as

$$L_{xe}(f(x,\theta),y) = \sum_{i=1}^{n} y_i \log(f(x_i,\theta)) \tag{4.2}$$

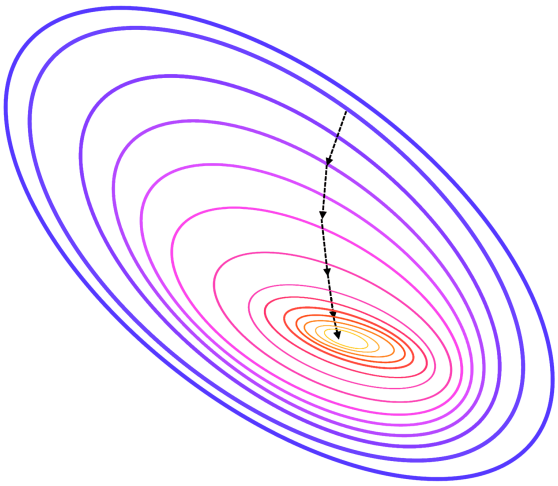


Figure 4.1: A demonstration of the gradient descent concept where the black arrows represent the small steps taken in the negative gradient direction of underlying function shown with the level curves.

where $f(x, \theta)$ is the distribution over hypotheses inferred by the DNN parametrized by θ, y represent the target distribution and n is the number of hypotheses. For batch input the overall cost function is defined as the expectation over the batch of data as described in Equation 4.1.

For regression tasks, one of the most popular cost functions is the mean squared error (MSE). This error function is both convex and symmetric which are desirable properties of a cost function because it then penalizes both positive and negative errors equally. The MSE is, however, rather sensitive to outliers since the error grows quadratically for linear deviations. The MSE for a single example x is defined as

$$L_{mse}(f(x, \theta), y) = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i, \theta))^2 \tag{4.3}$$

where $f(x, \theta)$ is the inferred quantities of the DNN, y represents the target values and n represent the number of quantities inferred by the DNN. The overall cost function for batches of data is similarly defined by Equation 4.1.

4.1.2 Gradient Descent

Gradient descent (GD) is an iterative optimization algorithm for which there exists different variations that introduce a trade-off between speed and accuracy of updates. The most crucial aspect which gives preference for one variation over another is the size of the total dataset available for training. This is the case because the difference between variations of this methods boils down to the amount of data used for each update.

The regular gradient descent algorithm makes use of the complete dataset at each update step, making each update very expensive in terms of time and computational

cost. Furthermore, using the complete dataset disables the possibility for continuous inclusion of data in an online fashion. However, the gradient is generally more accurate compared to methods using only a subset of the available data. In contrast, the completely stochastic version makes use of the opposite approach. At each update step, only a single data point is used. This provides means for fast updates, albeit not as accurate as regular GD because gradients are calculated with respect to single examples. The completely stochastic variation is thus particularly suited for online learning where new data periodically becomes available.

Lastly, a popular variation of GD is the so called mini-batch gradient descent which uses a subset of the data points, more than one but less than the complete dataset, for each update. This method introduces a tuning parameter, the mini-batch size, which effectively determines the trade-off between computational speed and accuracy of the gradients. The update rule is defined analogously for the three variations, with the difference being the use of a subset of data rather than a single example or the full dataset. The update rule is defined as

$$\theta \leftarrow \theta - \eta \nabla_\theta J(f(x, \theta), y), \tag{4.4}$$

where η represents the learning rate parameter and x and y represent either a single, a subset, or the complete set of data points. The learning rate is usually a small value in the range $(0, 1)$.

Deep neural networks are in general highly non-convex and rather complex functions. Due to this, a number of problematic behaviors can often be encountered. When such functions are to be optimized using stochastic optimization methods, a common problem known as ill conditioning might arise. When the parameter space of a DNN is a highly curved space (as is often the case), meaning that the second order derivatives are large, there is often a risk of taking to large update steps. This means that in order for a gradient descent update to lead to a smaller cost, the learning rate often need to be reduced. As such, if the parameter space is to highly curved, the learning rate might need to be reduced to the extent that it is close to zero which would result in the optimization getting extremely slow.

Other issues encountered during optimization, such as exploding or vanishing gradients, can often be encountered when the architecture of a neural network is very deep [15]. Feed forward networks are more prone to the vanishing gradient problem because a large number of multiplications are performed on such architectures. This often results in a diminishing value for the gradient, causing the optimization to slow down. Local minima and other flat regions, such as plateaus in the parameter space, pose another problem. At such points, the gradient is already effectively zero, causing the optimization to halt. Choosing good hyper-parameters is yet another, often difficult, task. As mentioned, the learning rate is an example of such a parameter that if selected too small will make the optimization process very slow and if selected too large might cause the optimization to diverge or make it unstable [19].

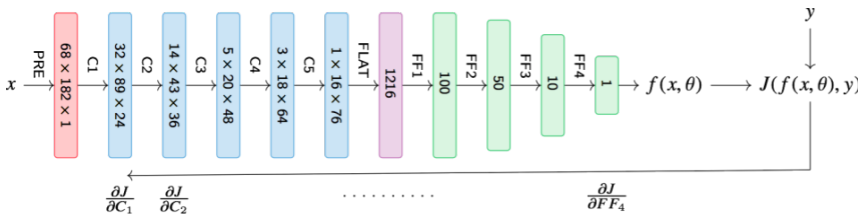


Figure 4.2: Illustration of the supervised learning procedure. Inputs x are used to calculate a forward pass by which a cost function J is used to determine the error between the model output $f(x, \theta)$ and the ground truth y. The partial derivatives of this error with respect to the model parameters θ is then calculated using backpropagation. These error derivatives are finally used to update the model parameters by means of gradient descent.

4.1.3 Supervised Learning

Training a DNN on samples of input data coupled with ground truth target values is commonly referred to as supervised learning since the model learns from examples. An efficient algorithm for employing supervised learning for feed forward networks is the so-called backpropagation algorithm. This algorithm provides an efficient way of calculating the partial derivatives needed to apply parameters updates using the method of gradient descent. As the structure of a feed forward network constitutes a composition of (possibly many) functions, backpropagation makes use of the chain rule of calculus to obtain the gradients w.r.t. the model's parameters.

The supervised learning process is an iterative process which alternates the following steps a pre-defined number of iterations or until the model parameters converge. The process is illustrated in Figure 4.2. Given a DNN model $f(x, \theta)$ parametrized by θ, input examples x and corresponding target values y, do

1. Calculate a "forward pass" $\hat{y} = f(x, \theta)$.
2. Calculate the gradient $\nabla_\theta J(\hat{y}, y)$ w.r.t. θ using backpropagation.
3. Update parameters $\theta \leftarrow \theta - \eta \nabla_\theta J(\hat{y}, y)$ using gradient descent.
4. Stop if convergence or pre-defined steps are reached, otherwise repeat from 1.

4.1.4 Dropout

The dropout method of regularization belongs to a class of techniques which in the ML context are known as ensemble methods. This class of methods aims at reducing generalization errors by means of model averaging and has been shown to be extremely powerful and reliable [15]. In a DL context, working with a large number of DNNs in parallel will quickly be unmanageable as a result of the large amount of parameters generally used by such models. To avoid this, the dropout method provides an approximation for averaging an exponential number of models which has been shown to be both powerful and computationally cheap for tasks in vision, speech and computational biology [20]. An additional strength of the dropout method is that it easily integrates with other regularization methods and

can be successfully used with an GD optimization scheme.

In a general feed forward network, the mathematical description of a forward pass of data through a layer l as described in Section 3.1.2 gives an output $a^{(l)}$. The dropout algorithm modifies this output during the training phase by multiplying each output unit $a_i^{(l)}$ of a layer with a random variable $r_i^{(l)} \sim \text{Bernoulli}(p)$ such that the output of that layer is replaced by

$$a^{(l)} \leftarrow a^{(l)} \odot r^{(l)} \tag{4.5}$$

where $r^{(l)}$ represents a tensor of independent Bernoulli random variables [20] and $\odot$ denotes element wise multiplication.

By applying this modification to each layer, samples of sub networks from a collection of 2^n possible configurations can be obtained where n represents the number of units in the base network architecture. All sub networks share the same weights so by applying this method of sampling, most configurations will only be used for training a small number of times or not at all depending the size of the base architecture. It has further been suggested that for most architectures and units, the probability p for which a unit is dropped, can be fixed and set to 0.5 (or determined using a validation data set) to achieve good regularization [20].

4.2 Implementations

In this section, a description of the important parts of the training procedure will be given. The Tensorflow library, a tool developed by Google for efficient computation and visualization of mathematical graphs (which is extensively used in this project), will be briefly presented. Then a description of the data and input pipeline is given, focusing on how it was implemented in order to reduce input lag from the hard drive and to make the datasets more manageable. Further, an overview of the hyper parameters and error metric used for training of the models are shown. Finally, the process of incorporating the trained models in the simulated environments is explained.

4.2.1 TensorFlow

TensorFlow is an open source software library for machine intelligence, initially developed by the Google Brain Team [21]. The library supports definition of computational graphs supporting multidimensional data arrays (tensors). TensorFlow was initially developed for conducting research on large scale deep neural networks but is very general purpose. Any machine learning algorithm that can be represented in the form of a computational graph can be implemented using this framework. TensorFlow supports computation on a variety of hardware, including both single and multiple CPU/GPU clusters. Further, TensorFlow provides APIs in several programming languages for both construction and execution of TensorFlow graphs. The most complete and easy to use API is written in the Python programming language, the core API is written in C++ and other experimental API's include programming languages such as Java and Go.

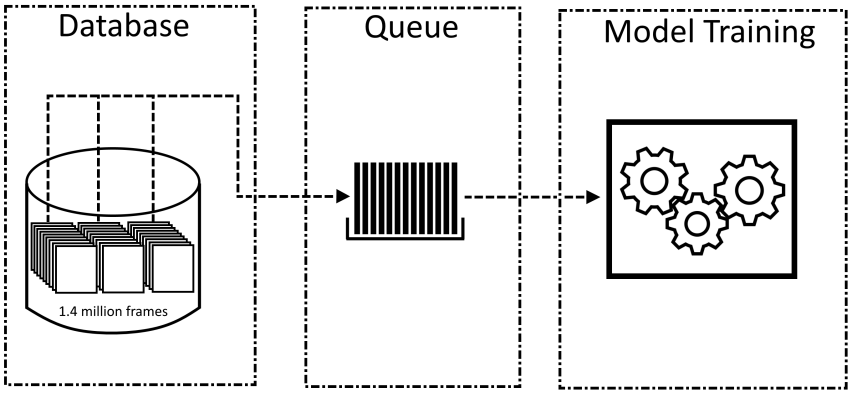


Figure 4.3: A schematic overview of the input pipeline for model training. The high latency of accessing the hard drive where the training images are stored is mitigated by running a separate thread, filling intermediate queues in RAM which the model training thread loads its data from.

4.2.2 Datasets and Input Pipeline

There are several ways in which one can feed data into models created with the TensorFlow environment. One way is to train models using one batch of data at a time, loaded from the hard drive as needed. This method works well for data sets that are quite small as the accumulated overhead from accessing the hard drive is limited over a short training session. For moderately sized data sets, the overhead might be a bit cumbersome. One way to remedy this is to load most of the data into RAM, requiring only a single large load time, but at the cost of a larger memory footprint. However, for very large data sets, this might not be possible as few ordinary computers have more than 64 GB of RAM.

Another option is to only load in parts of the data at a time. For this work, the datasets were initially converted into TensorFlow TFRecord format in order to leverage data queues. Furthermore, packaging the data into TFRecords had the benefit of combining the very large amount of separate training examples to larger, more manageable files.

For training, the queues were then populated with data from background threads, ensuring that the training procedure, running on the main thread, was protected from disk lag. From the TFRecords, training examples (image-steering pairs) were loaded into the queue in batches. Batches were formed by shuffling examples in random order from random files, ensuring that the model observations become as uncorrelated with each other as possible. This enabled a more smooth and reliable training procedure in terms of performance and time consumption. A schematic representation of this procedure can be seen in Figure 4.3.

4.2.3 Model Parametrization and Training

Three different versions of the steering model were trained, one for each of the datasets described in Chapter 2. For these three versions, the same specification for the training procedure was used. The training was performed in a supervised

setting, with the mini batch size set to 64 images at a time. The batches of images were fed into the model which then predicted some outputs. These outputs were then compared to the ground truth for those images (the recorded SWA) and the mean squared error loss was calculated.

The actual training, or optimization of the network, was performed using gradient descent with the Adam [18] optimizer with an initial learning rate of 0.0001. Further, in order to improve the networks generalization, dropout was applied of the first three fully connected layers with a keep probability $p = 0.5$. The probability for dropping units at each layer was selected because it has been suggested that this value of p is close to optimal for a wide variety of architectures and tasks [20].

Because the difference in data size for the training of the three different models was quite significant, the total number of batches (iterations) each was trained on differed substantially as well. For the model trained on Volvo expedition data, roughly 200 thousand iterations seemed to produce the most general model. For the models trained on Unity and CarMaker data, roughly 20 thousand training iterations seemed to produce the best results.

4.2.4 Simulator Interfacing

For the purpose of evaluation in a closed loop setting, the developed models needed to be integrated with the simulation environments described in Chapter 2. The models had to receive information from the simulation and send back the inferred actions in order to achieve real time decision making.

For this end, the TensorFlow models were converted to a platform independent format (protocol buffer format), able to interface most software languages. The interface with the Unity game engine did not explicitly require this conversion since the interface was based on the Python programming language. Integration with the CarMaker environment however, required the use of the C and C++ programming languages, and thus required this platform independent format.

The initial version of the Unity simulator was adapted by the Udacity organization [22] from the Unity standard assets to serve as a platform for one of their "nanodegree" programmes. The communication interface between the Unity game engine and the model was done via a websocket interface, where images from Unity were sent to the model for processing and the output of the model transmitted back to control the virtual car. For every rendered frame, the captured image was transmitted from the virtual front facing camera to the model which then transmitted back the inferred steering action.

For their programme, Udacity had created a small python script to serve as the interface between their model and the Unity engine. This script was substantially modified in order to utilize the model developed in this work. Both the steering and the vehicle throttle was controlled via this script, and as the created model only gave a steering command as output, a very simple proportional controller was created to generate the acceleration signal. Finally, support for converting the curvature $\frac{1}{r}$ measure output from the model to a Unity compliant format was created. The transformation was based on measurements of the virtual vehicle made via the Unity editor.

To interface the CarMaker environment, a modified user project was created. A user project is the equivalent of an internal user application, with user accessible C code. From this application, in principle all aspects of the simulation loop can be controlled, for example assignment of the steering wheel angle of a simulated vehicle. Because the application code can be accessed directly, integration of custom C/C++ code for vehicle control was somewhat straight forward. In order to control the steering of a simulated vehicle in CarMaker, a custom C++ library based on the TensorFlow source code was built. This library acted as an interface between CarMaker and the steering model developed in this thesis. In CarMaker, the state of the simulated environment was updated from a main control loop where the state update included all actions taken by actors in the environment. For example, updating the position of a simulated vehicle driving at a certain velocity with a particular angle of the steering wheel.

To obtain front view images from the simulated vehicle, an extension to the CarMaker software needed to be used. This was the so-called video data stream (VDS) extension that made it possible to include a virtual front view camera into the simulated environment. The main control loop and the VDS extension (image feed) was not synchronized and could be viewed as two separate applications running in parallel. This is much like in a real world setting where the environment continues to change and does not wait for a driver to process received visual input and to act on it before transitioning to the next state.

For simulations, when an image was received from the virtual camera, it was passed to the model using the C++ library and was used to compute the desired steering action to take. The steering action was then sent back using the C++ interface and the state of the vehicle was updated.

5
Model Evaluation

In many cases, it can be almost impossible to determine best options by visual inspection. So instead we often use some kind of metric. Usually it takes the form of some number, discriminating the wheat from the chaff. Unfortunately, reality is subjective. Difference in perspective might play a significant role in how objects and phenomena are perceived. Due to the difference in perspective, deciding on which metric to use is not a straightforward process.

In this chapter, the concepts of closed loop simulation and transfer learning are initially introduced in Section 5.1 and 5.2. Focus is put on the domain adaptation aspect since (the primary) model in this work was trained using real world data but evaluated in a simulated environment. An initial view of the model's predictive capacity is given in Section 5.3 where closed loop tests are demonstrated. Section 5.4 briefly outlines what the model seems to pay attention to in the different domains. Then, in Section 5.5, the proposed penalty metrics for driving performance are defined. Using these performance metrics, the model behavior on realistic road representations is investigated in Section 5.6 where the model's predictive capacity is analyzed, compared, and discussed for different cases of driving.

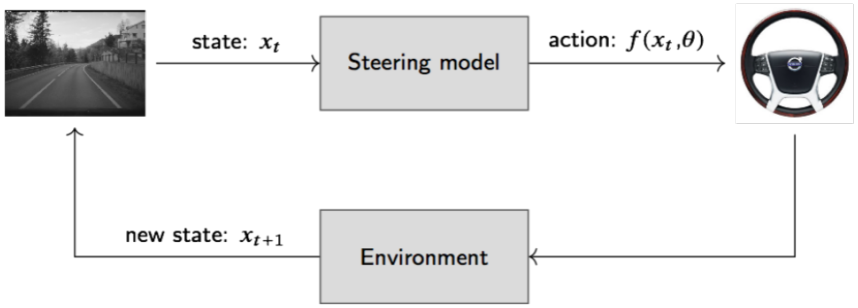


Figure 5.1: Schematic illustration of the closed loop simulation setup. At each point in time, the model observes a state/observation x_t and decides on what action $f(x_t, \theta)$ to take. The action is realized in the simulated environment which is updated to a new state x_{t+1}, observable by the model.

5.1 Closed Loop Simulation

For the purpose of this work, different models were trained to imitate the behavior of human drivers in order to learn the concept of steering a vehicle. As described in Chapter 4, the models were trained on the various datasets described in Chapter 2. This was an iterative process where models with specific training setup, data selection and configurations were subject to evaluation. Based on the evaluation results, the model, learning procedure and data were updated in order improve performance.

These types of models are commonly evaluated using validation datasets. That is, data not seen by the models during the learning phase annotated with ground truth labels. However, as is the case when steering of a vehicle, there are many valid ways in which one can steer at any given point in time. This is a sequential decision-making problem. Therefore, requiring that a model predicts the exact same action as that of a particular human driver is not necessary. For example, given a road segment of a curve, it is most likely the case that it can be maneuvered correctly by a large variety of different steering actions, other than those taken by a particular human driver. To this end, evaluation in a closed loop setting is a way of empirically determining the stability of a system. Figure 5.1 illustrates the closed loop control setup where at each point in time a state/observation x_t is seen by the controller/model. The actions taken by the model at each point in time will thus influence the next state x_{t+1} and subsequently the behavior of the model at future points in time.

5.2 Transfer Learning

Creating a system that works satisfactory in a real-world setting based on synthetic data is a desirable goal. This would reduce the need for a lot of expensive and often time-consuming data collection processes. However, if large databases already exist (as is often the case for large car manufacturers) or are easily obtained, an equally desirable goal is to make good use of the available data. If data driven models could be trained using available real-world data while still being able to be tested and evaluated in simulated environments, this would in the same way reduce the need

for costly HIL and/or vehicle tests at early stages of development.

The capabilities of the models used in this work were analysed in terms of knowledge transfer across different domains, in this particular case, training the model on real data but evaluating on synthetic data. This refers to the method of transferring knowledge from a known set of conditions to a new set of conditions. Transfer learning is defined in [23] as knowledge transfer between domains $D_1, \ldots, D_n$ and/or tasks $T_1, \ldots, T_n$. A domain D is defined by a feature space $\mathbb{X}$ and a marginal probability distribution $P(X)$ for $X = x_1, \ldots, x_n \in \mathbb{X}$ and is denoted

$$D = \{\mathbb{X}, P(X)\}. \tag{5.1}$$

In the context of this work, the feature space $\mathbb{X}$ corresponds to the space of all possible image representations (the pixel space) and the random variable X then represents a particular image.

A task T is similarly defined for a given domain D by a target space $\mathbb{Y}$ and a conditional probability distribution $P(Y|X)$ denoted

$$T = \{\mathbb{Y}, P(Y|X)\}. \tag{5.2}$$

Analogously, in the context of this work, $\mathbb{Y}$ represents the space of all possible steering actions, where the random variable $Y \in \mathbb{Y}$ is a particular steering action. The conditional probability distribution thus describes the probability of a given steering action conditioned on a given image and is what is learnt using training examples $y_i \in Y$, $x_i \in X$.

Pan and Yang [23] divides the possible applications of TL into four categories based on knowledge transfer between source and target domains (D_S, D_T) and source and target tasks (T_S, T_T). The category which is of the most relevance for this work is known as domain adaptation and occurs when the source and target domains have different marginal probability distributions $P(X_S) \neq P(X_T)$. The remainder of the categories consists of source and target domains having different features spaces ($\mathbb{X}_\mathbb{S} \neq \mathbb{X}_\mathbb{T}$) and source and target tasks having different target distributions ($\mathbb{Y}_\mathbb{S} \neq \mathbb{Y}_\mathbb{T}$) and/or conditional probability distributions ($P(Y_S|X_S) \neq P(Y_T|X_T)$). The last two cases often arise in the case of task adaptation and is probably the most common case of transfer learning applied to deep neural networks, in particular to CNNs. For classification tasks, CNNs are usually pre-trained on some large database and the feature representations obtained are subsequently used for different classification tasks.

For this work, domain adaptation occurs since training was done in a source domain that consisted of real world images and the model was subsequently applied to the target domain of synthetic image data for evaluation. The feature spaces are thus the same (the pixel space) while objects and environments generally look different. Intuitively, this is the case because the probability of observing a real-world image in the simulated environment is (extremely) small. Figure 5.2 illustrates the different domains used for training and evaluation.

Through the remainder of this chapter, the results presented show that the proposed type of model indeed works in a satisfactory manner when domain changes are applied. The results therefore suggest that pre-existing real data can be used for both rapid and inexpensive development of data driven models.

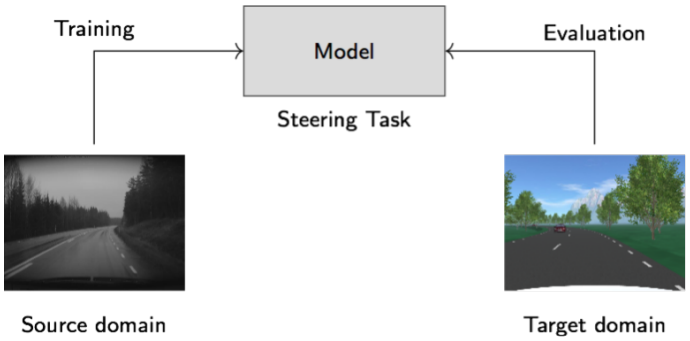


Figure 5.2: Schematic illustration of the instance of domain adaptation applied to the model used in this work. The model is trained on real world image data from the source domain to perform the steering task and subsequently applied to the target domain of synthetic image data for evaluation.

5.3 Predicting Steering Actions

Based on the synthetic data gathered from the Unity and the CarMaker environments, two different models were created to serve as a proof of concept, as described in Section 4.2.3. These models were trained on the same type of data that they would be exposed to during evaluation (no domain adaptation) and could therefore serve to indicate whether the model architecture even had the *capacity* to learn the concept of driving. Had these models failed, it would have served as a good indication that another model architecture might have been required.

As mentioned in Section 2.4 and 2.5, the datasets these models were trained on was significantly smaller than that of the Volvo expedition dataset and thus required only a fraction of the time for training. Naturally, using only a very small dataset meant that the models were likely to become quite limited in terms of generality, but since their purpose was to demonstrate the *capability* of learning, this was not considered an issue.

Both of the proof of concept models were evaluated empirically in the simulator environments, where their performance could easily be assessed. Recordings[1] of the simulations clearly shows that the models have learned the concept of following a road quite nicely, based on a small amount of data. The model trained on the 1.4 million images from the Volvo expeditions had a dataset roughly 28 times greater than the proof of concept models, and therefore it was allotted significantly more time for its training phase. However, unlike the models trained on synthetic data, this one was to endure a domain change for its evaluation. Similarly as for the proof of concept models, the Volvo expedition data model was to be evaluated in both Unity and CarMaker as well. However, for this model, rather than just performing very generic tests, the initial performance of the model was evaluated based on few specifically constructed scenarios.

Because ordinary driving can be decomposed into the three different scenarios, driving straight, turning left and turning right, these three scenarios were considered

[1] https://youtu.be/jC8FFt-bV_s
https://youtu.be/IHgpG3OlGbg

Figure 5.3: The Unity sky-car driving in a right turn on a 6-lane highway in the Unity game engine.

good base cases to investigate the effectiveness of the new model. Based on the performance on the base cases, conclusions as to whether the domain change had affected the model performance could be gauged. In addition, performing these tests might also have revealed any latent bias in the training data.

In this section, the performance on the base cases will be presented for each of the two simulation environments. Even though some effort was made in order to make the two simulation environments as similar as possible in terms of road geometries, Unity lacked the support CarMaker provided for easily specifying roads. A such, some mismatch between the two was unavoidable. Thus, a detailed description of the specification for the base case road geometries will be given for each respective simulation environment. Nevertheless, the results from tests will be presented, coupled with a discussion of the nature of the model's predictive capability.

Finally, this section ends with an empirical robustness analysis of the model. The robustness of the model was analysed based on how well it handled sudden disturbances in the form of fault injections. In this case, the fault injections took the form of temporary overrides to the model output, causing a misalignment of the vehicle in relation to its lane. From this erroneous state, the model's capacity for recovery was investigated, and the results discussed.

5.3.1 Driving in the Unity Game Engine

In the Unity environment, samples of the inferred steering actions (curvature) were collected for specific driving scenarios by closed loop simulation using the model trained on Volvo expedition data. The road segments used for simulation consisted of parts of a 6-lane highway from the Kajaman road package [24]. The scaling of this road was set such that each lane had a width of 3.9m, a roadside width of 0.6m foreclosed by barriers of height 0.8m. The center, left most and right most lane markings were solid lines while markings between lanes were 3.3m line segments with 3.6m spacing.

The road geometries used for evaluation corresponded to initial stretches of roughly straight road followed by either a left or right turn (approximately 90°)

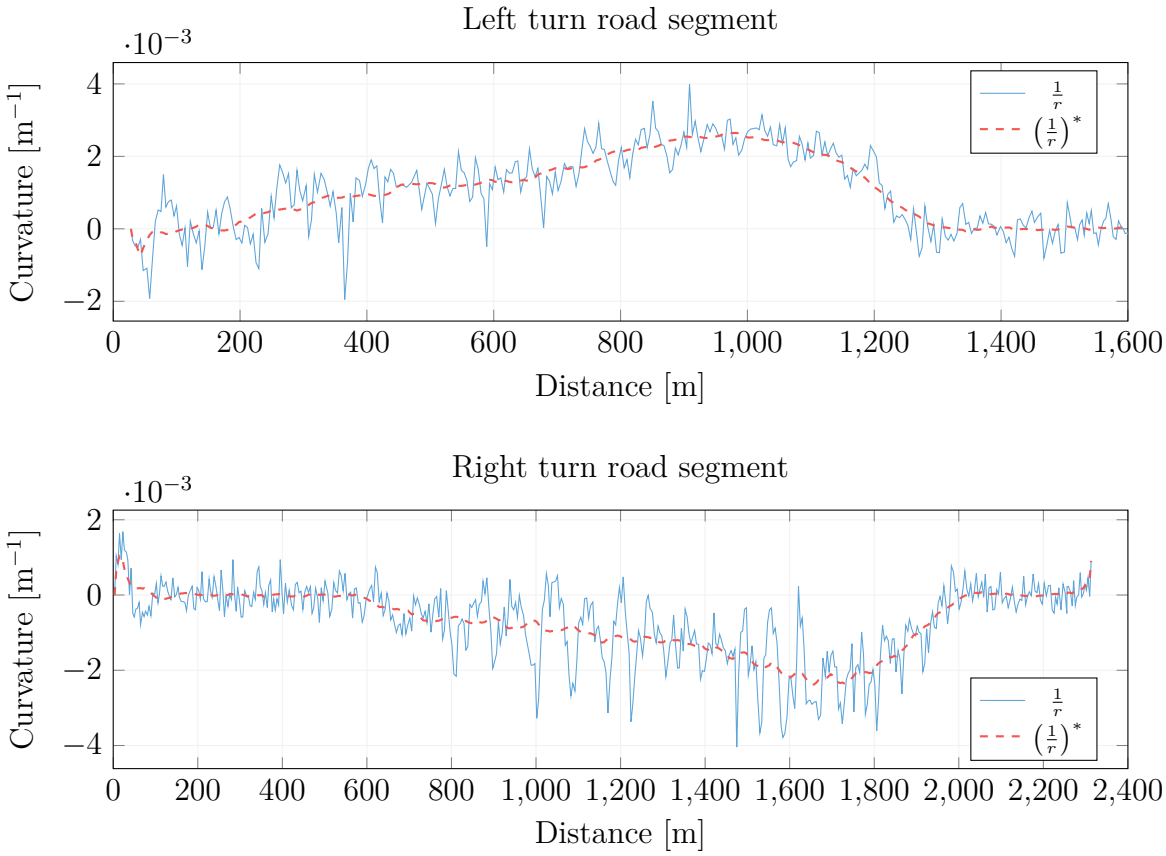


Figure 5.4: Predicted curvature values $(\frac{1}{r})$ and smoothed curvature values $\left(\frac{1}{r}\right)^*$ from the steering model introduced in Section 3.2. The model was trained on Volvo expedition data and samples were collected from two evaluation runs in the Unity game engine on a 6-lane highway as can be seen in Figure 5.3.

Figure 5.5: Volvo XC90 driving in a left turn on a 6-lane highway in the IPG CarMaker environment.

with varying radii. The radii were initially large and decreased towards the end, i.e. the curves got sharper towards their end. The total driving distance was approximately 1.7–2,4km. Figure 5.3 illustrates a third person view of the initial part of the right turn road segment, where the Unity sky-car is positioned in a center lane. Recordings[2] of the driving simulations with the vehicle speed set to a constant 100 km/h show the model performance.

The predicted curvature values for both the left and the right turn simulations can be seen in Figure 5.4. In addition to the predicted curvature, a smoothed average of the signal is included in order to observe the general trend of the predictions. As can be seen, the predicted signals are rather noisy and shifts quite a lot between samples. This is not surprising because the steering model predicts the curvature on a frame–by–frame basis. Therefore, it seems to alternate between over and under estimating the actions to take for an optimal path.

From empirical observations of these driving scenarios, the vehicle stays rather well positioned in its lane and does not oscillate within the lane as much as the curves in Figure 5.4 seems to suggest. In summary, this model seems to have adapted rather well to the Unity environment even though it never observed any data from that domain during training.

5.3.2 Driving in IPG CarMaker

In the IPG CarMaker environment, samples of the inferred steering actions (curvature) were similarly collected for specific driving scenarios as a basis for analysis. The road segments used in this case also consisted of a 6-lane highway where each lane had a width equal to 3.75m and a roadside width of 0.5m. This lane specification corresponds to a road of "high standard" in Sweden [25]. Lane markings are similar to those of the road used in the Unity environment. The center, left most and right most lane markings were solid lines while markings between lanes were 3m line segments with a 9m spacing. The road geometries consisted of approximately

[2]https://youtu.be/DE1q1qPoJ64
https://youtu.be/B98VdNiNBqo

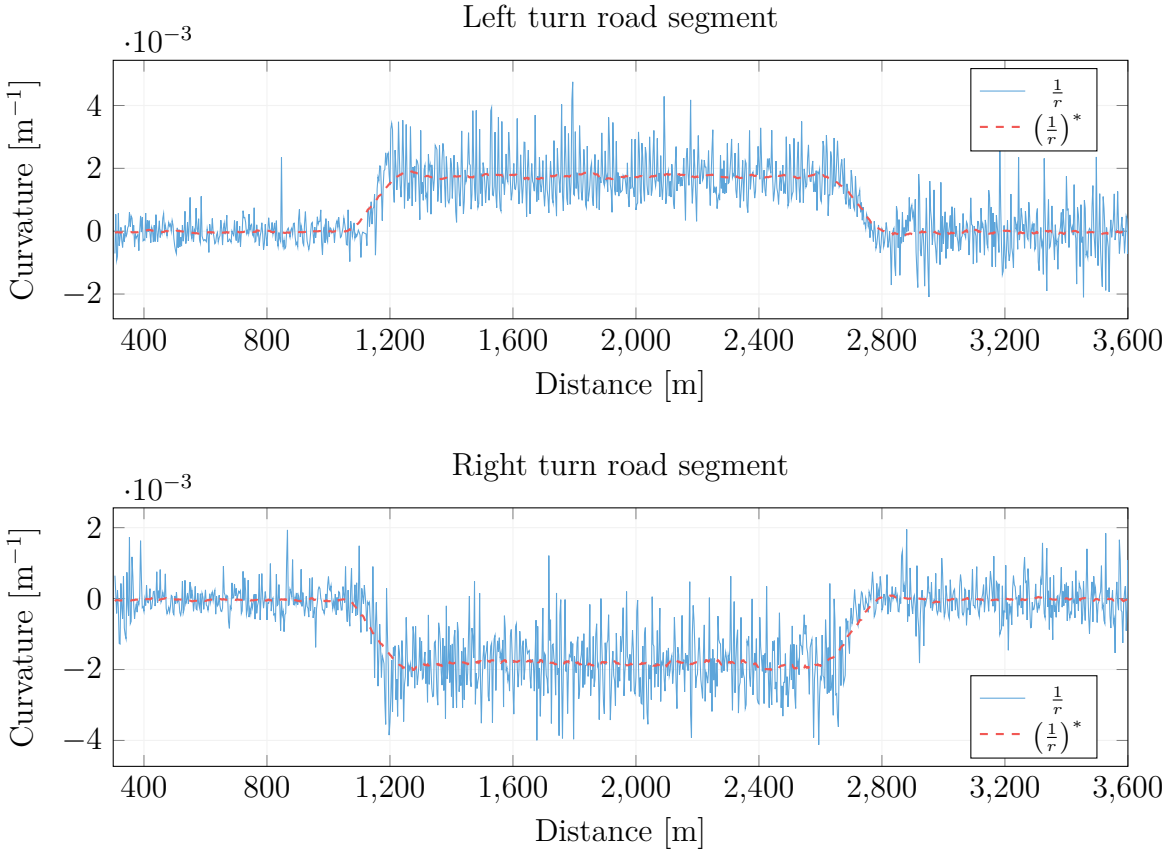


Figure 5.6: Predicted curvature values $(\frac{1}{r})$ and smoothed curvature values $\left(\frac{1}{r}\right)^*$ from the steering model introduced in Section 3.2. The model was trained on Volvo expedition data and samples were collected from two evaluation runs in IPG CarMaker on a 6-lane highway as can be seen in Figure 5.5.

1km straight road followed by either a left or right 90° turn with a constant radius of 1km and ended with additional stretches of straight road. Figure 5.5 illustrates a third person view of the left turn road segment where a simulated Volvo XC90 is positioned in a center lane.

Recordings[3] of the driving simulations performed in CarMaker, with a fixed speed of 100 km/h, demonstrates the model performance. The predicted curvature values for both the left turn and the right turn simulations can be seen in Figure 5.6 in addition to the corresponding smoothed predictions. Similar behavior as that observed from simulation in the Unity environment can be seen in this case. The predicted values seem to alternate between under and over estimating the curvature. By similar empirical observation of the driving scenarios as in the Unity case, the vehicle stays rather well positioned in its lane and does not show any large oscillatory behavior. In summary, this model seems to also adapt well to the CarMaker environment even though it never observed any data from that domain during training.

[3]https://youtu.be/IF6oV9et9T0
https://youtu.be/STluSPG3V2g

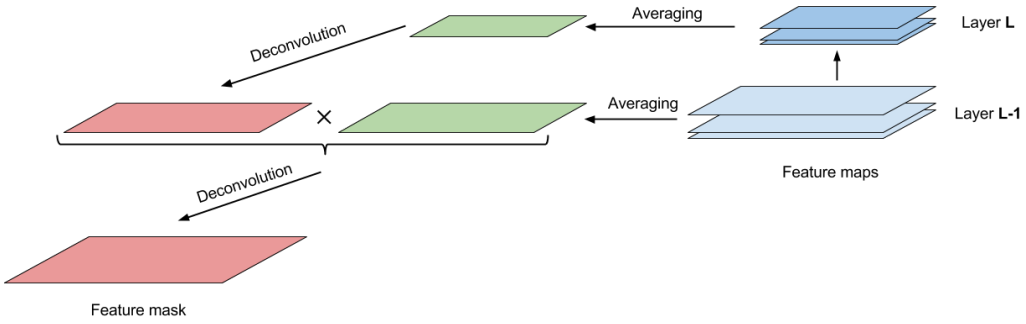


Figure 5.7: Illustration of the visual backpropagation procedure. The averaged feature map at a layer L is deconvolved (up sampled) and elementwise multiplied with the average feature map of the previous layer $L-1$, resulting in an intermediate feature mask. The intermediate mask is then deconvolved to produce a feature mask for layer $L-2$ and the process is repeated until the input layer is reached.

5.3.3 Robustness

When designing a control system, one usually intends it to be run under some nominal conditions. During these conditions one would like the controller to behave in such a way that it fulfills some set specification. In this case for example, one can consider the specifications being to stay properly aligned in a lane for extended periods of time. To handle this, the controller would have to be able to handle the slow changes in road geometry one encounters while driving on highways. Of course, a proper control system should be able to handle the nominal cases but this is not always sufficient. Sudden changes in the environment or changes in setpoints, may force the system outside of its nominal range. In such cases one would like the controller to have such characteristics that it manages to reinstate the nominal behavior again rather than having instability ensue.

In this case, the robustness of the created model for steering the vehicle was empirically analysed. The model was set to run on the road segments defined in Secion 5.3.2 and 5.3.1 but at some random point in time, the model would be subject to a fault injection. Whenever such an event occurred, the steering of the vehicle was overridden as to force its front outside of the currently traversed lane. From this new orientation, control was reinstated to the model which then had to correct for the error[4]. These correspond to steering angle offsets to the left and to the right for both simulation environments respectively. From the simulations, they clearly show that the model exhibits some characteristics of robustness. The model manages to guide the vehicle back into the lane after having the vehicle's position displaced.

5.4 Domain Invariant Features

To further investigate if a model trained on real world images generalizes across different domains, an analysis on the decision-making process was made. This analysis was based on visual backpropagation [26] which is an algorithm for determining, in

[4]https://youtu.be/alP_zydEVac

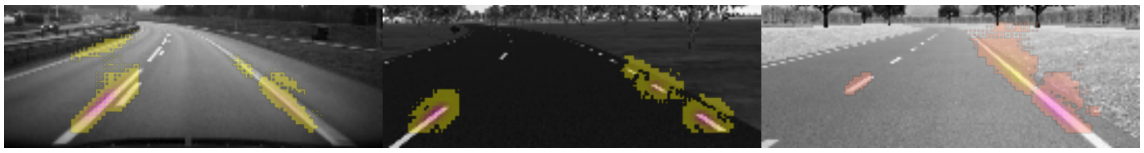


Figure 5.8: Illustration of the most significant pixel regions (pixels with the highest activation values), obtained through visual backpropagation from the proposed model trained on Volvo expedition data. Images are samples from three different environments that correspond to a Volvo expedition (left), IPG CarMaker (center), and the Unity game engine (right).

the case of image data, the pixel regions of an input image that have the highest activation for a given model output. The purpose of this analysis was to determine if these regions were similar for different domains, i.e. similar when images from the simulation environments were fed to the model trained on real image data. The visual backpropagation algorithm illustrated in Figure 5.7 has been summarized in [27] and is repeated here for clarity.

1. In each convolutional layer, the feature maps are averaged.
2. The final averaged map is scaled up to the size of the map of the layer below using deconvolution. The kernel size and stride used for the deconvolution are the same as in the convolutional layer used to generate the map. The weights for deconvolution are set to 1.0 and biases are set to 0.0.
3. The up-scaled averaged map from a deeper level is then multiplied with the averaged map from the layer below (both are now the same size). The result is an intermediate mask.
4. The intermediate mask is scaled up to the size of the maps of layer below in the same way as described Step 2.
5. The up-scaled intermediate map is again multiplied with the averaged map from the layer below (both are now the same size). Thus, a new intermediate mask is obtained.
6. Steps 4 and 5 above are repeated until the input is reached.

For this case, samples of images from Volvo expeditions, Unity and IPG CarMaker were processed by the model trained on Volvo expedition data. The feature maps obtained from the processing stage was then subject to visual backpropagation.

Figure 5.8 illustrates one result showing a strong indication that lane markings seems to be important features for making steering decisions, which has previously been argued for in [26]. The left most image is a sample from the Volvo expedition data and as can be seen, the pixel regions with the highest activations for both the center image (obtained from IPG CarMaker) and the right most image (obtained from Unity) are highly similar to that of the real-world image. Based on the similarity in activations, this shows that knowledge obtained by learning from real world image data is transferable across domains.

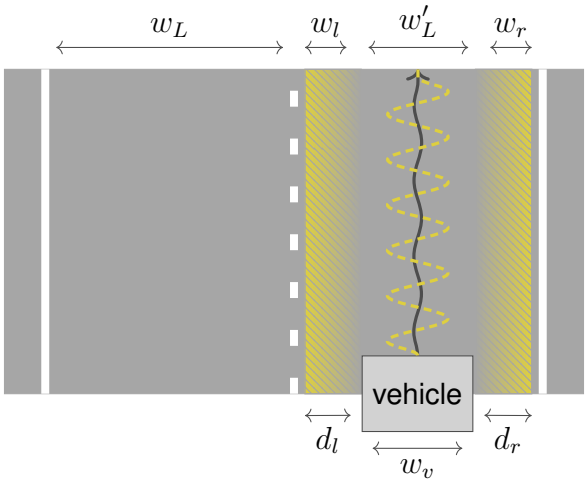


Figure 5.9: Illustration of a road segment showing the lane width w_L, penalty regions w_l and w_r, non-penalized lane region $w_L^{'}$, vehicle width w_v and vehicle–to–lane marking distances d_l and d_r.

5.5 Performance Metrics

Rating the performance of a driving algorithm is not a trivial task. The performance of driving needs to be quantified based on measures representing what good vs. bad driving behavior is, which is subjective to some degree. However, in this section, two different penalty metrics for assessing the quality of driving are proposed which can be used for comparing driving algorithms. Firstly, a metric penalizing erroneous vehicle positioning is introduced and is followed by the description of a metric with the purpose of penalizing noisy driving trajectories and reward low energy utilization in terms of vehicle steering.

5.5.1 Lane Positioning Penalty

A vehicle's position on the road, or more specifically in a lane, is an important measure of the driving performance. If an autonomous vehicle is able to stay well positioned within the regions for which it is allowed to drive, its driving performance with respect to positioning is said to be good, and the opposite if it is not able to stay within this region. In this section, such a measure is defined as an error metric denoted the lane positioning penalty.

Figure 5.9 illustrates a road segment with two lanes, both of width w_L, and a vehicle of width w_v positioned in the right lane. The positioning penalty regions of the right lane are defined as the widths of the striped areas on either side of the vehicle, denoted w_l and w_r for the left and right regions respectively. The width of the non-penalized region of the lane is thus defined as $w'_L = w_L - (w_l + w_r)$. Further, the distances from the edges of the vehicle to the left and right lane markings are denoted d_l and d_r respectively.

The lane positioning penalty $e_w(t)$ for time step t is analogously defined for the left and right regions. The penalty region widths are hyper parameters that

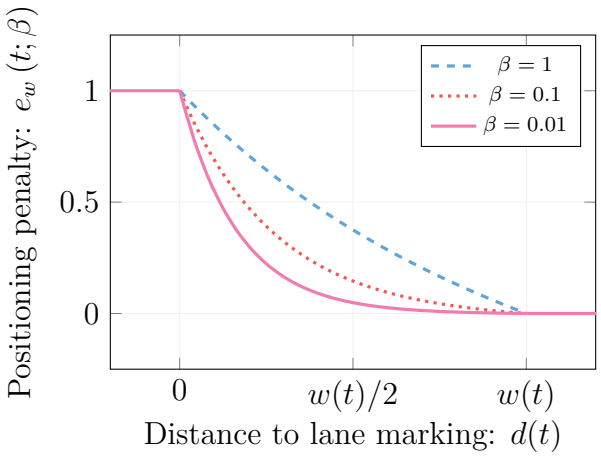


Figure 5.10: Lane positioning error as a function of vehicle–to–lane marking distance $d(t)$ with respect to penalty region $w(t)$ and β for different choices of β.

potentially vary over time and need to be tuned to represent erroneous positioning for different roads. However, given a defined penalty region $w(t)$ (either left or right) and vehicle to lane marking distance $d(t)$ (corresponding left or right), the penalty function is defined as

$$e_w(t;\beta) = \begin{cases} 1 & \text{if } d(t) < 0 \\ (\beta w(t))^{\frac{d(t)}{w(t)}} - \beta d(t) & \text{if } 0 \leq d(t) \leq w(t) \\ 0 & \text{if } d(t) > w(t) \end{cases} \tag{5.3}$$

where β is an additional hyper parameter that determines the shape of the error function w.r.t. the width of the penalty region $w(t)$. Figure 5.10 illustrates the positioning penalty as a function of the vehicle to lane marking distance $d(t)$ for three values of β. As can be seen, a value of β close to 1 corresponds to an almost linear increase in error for decreasing distance to the lane marking while a smaller value of β corresponds to an exponential increase in error when the vehicle gets closer to the lane marking.

The lane positioning penalty for a driving session (during $t = 1, \ldots, T$) is further defined as the average of the left and right positioning penalties for all time steps as

$$E_w(\beta) = \frac{1}{T} \sum_{t=1}^{T} \left[e_{w_l}(t;\beta) + e_{w_r}(t;\beta) \right]. \tag{5.4}$$

where $E_w(\beta) \in [0, 1]$ and 1 corresponds to entirely bad positioning and 0 corresponds to perfect positioning.

5.5.2 Accelerations and Jerks

In addition to vehicle positioning on the road, good driving behavior in the sense of how smooth and comfortable a driven trajectory is, can be characterized by the level of accelerations and jerks a vehicle is being subjected to. Consequently, since only lateral positioning (steering) is being controlled for this work, only lateral

accelerations and jerks will be used for comparing the level of comfort of driving tests.

Two trajectories are illustrated in Figure 5.9, one as a solid line and the other as a dashed line in the right lane segment of the road. The solid line represents the better of the two trajectories where the vehicle is subject to less lateral accelerations and jerks than from the other. Unfortunately, quantifying how bad a jerking driving trajectory is, is quite subjective. What some drivers experience to be uncomfortable might be acceptable for others. Furthermore, it could be a significant difference between what drivers and passengers consider acceptable. Because the passengers do not have direct control of the vehicle, they might be more easily startled by sudden changes in velocity etc. Based on the fact that all persons in an autonomous vehicle can be considered passengers, the reaction from them is more informative. From empirical studies discussed in [28], one can draw the conclusion that up to some certain threshold, jerks and accelerations only pose minuscule decreases in comfort. However, passing this threshold, the experienced discomfort rapidly increases.

This level of discomfort (or error) e_{lat} as a result of the acting acceleration and jerks is defined in [28] as

$$e_{\text{lat}}(t; g) = \begin{cases} \frac{x(t)^2}{g^2} & \text{if } x(t) < g \\ \left(\frac{5}{6} + \frac{x(t)^2}{6g^2}\right)^6 & \text{if } x(t) \geq g \end{cases} \tag{5.5}$$

where $x(t)$ is either the measured lateral acceleration (or jerk) for time step t and g the comfort threshold. In this case, the comfort threshold was set to 1.8m/s^2 (or 1.8m/s^3) as it was deemed a good value based on empirical studies performed on highways in Canada and China [29, 30]. The level of discomfort is illustrated in Figure 5.11 for this value of the comfort threshold. Based on this criterion, it is then possible to estimate the experienced comfort for a passenger for a particular driving scenario. Different agents driving the same scenario, with the same vehicle, can then also be compared in terms of comfort level.

The level of discomfort for a driving session (during $t = 1, \ldots, T$) is further defined as the average accumulated discomfort for all time steps as

$$E_{\text{lat}}(t, g) = \frac{1}{T} \sum_{t=1}^{T} e_{\text{lat}}(t; g). \tag{5.6}$$

5.6 Realistic Road Driving

Based on the proposed performance metrics, quantitative evaluation of the model could be made. In contrast to the tests described in Section 5.3.1 and 5.3.2, where the model had been subject to simple road segments, a more realistic evaluation was performed as well. For this case, real world road geometries, obtained via Google Maps, were included in CarMaker. For the Unity environment, a model of a highway was downloaded. These roads geometries were much longer and more realistic than the simple road segments from previous tests which meant that if the model accumulated some error over time, it should at some point have driven of the road.

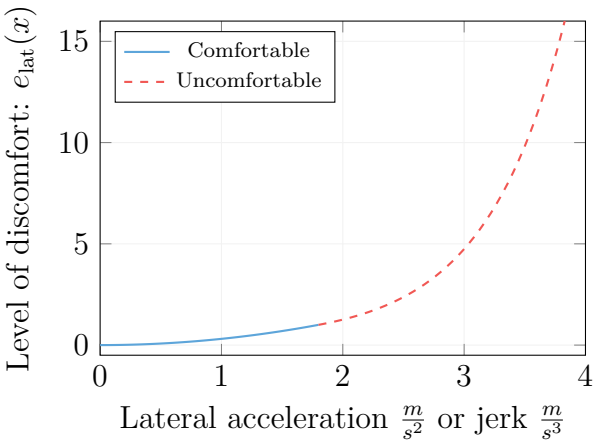


Figure 5.11: Level of discomfort metric $e_{\text{lat}}(x)$ for a comfort threshold $g = 1.8$ m/s^2 or m/s^3 for x being either lateral acceleration or jerk.

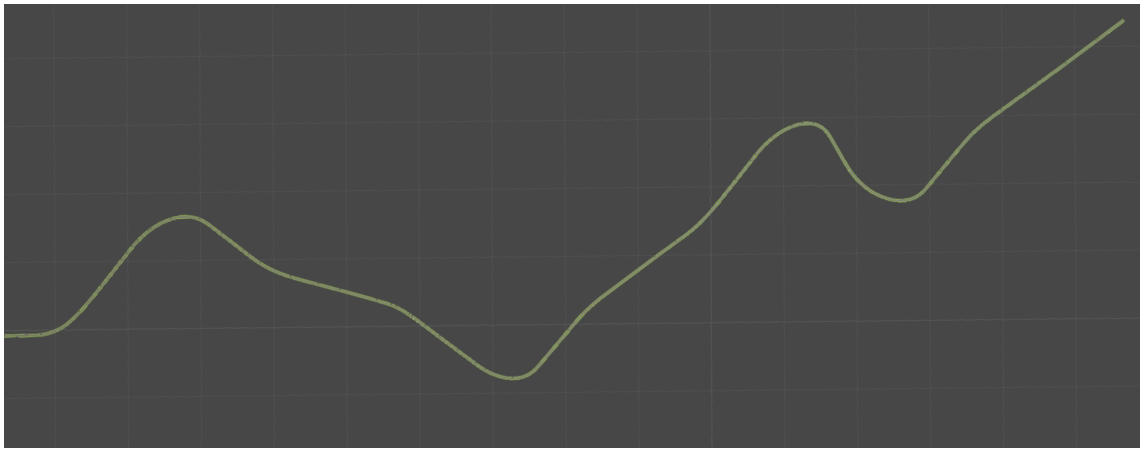


Figure 5.12: Birds eye view of the Kajaman highway road geometry used for evaluation in the Unity game engine. The complete road amounts to roughly 20km.

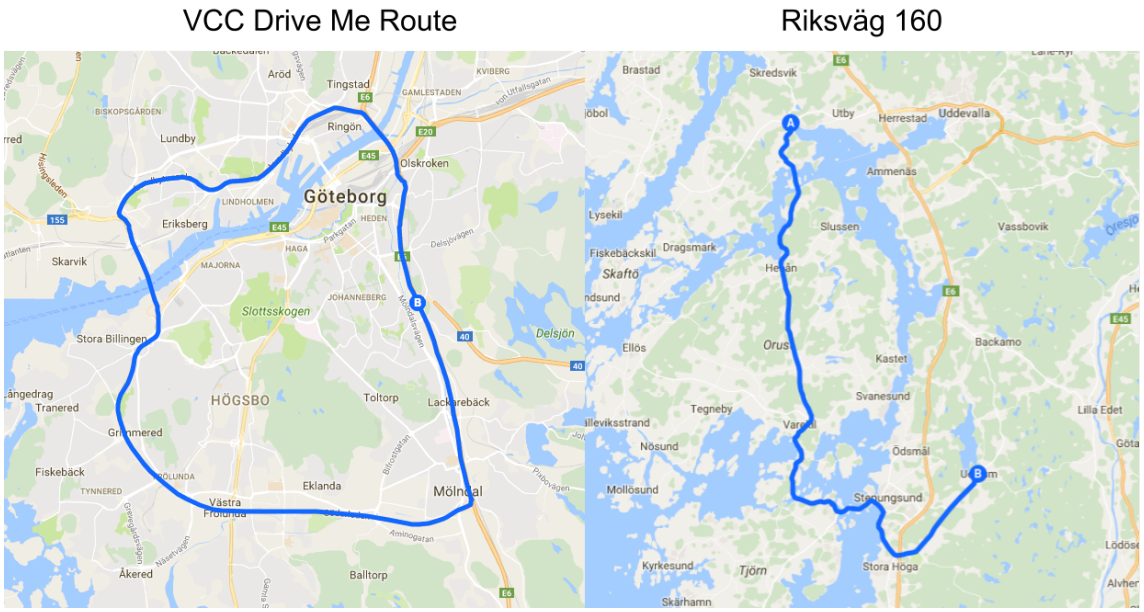


Figure 5.13: Illustration of two road segments used for evaluation in IPG CarMaker. These are 30km of the VCC Drive Me rout (left) and 53km of Riksväg 160 (right).

The model trained on Volvo expedition data was exposed to three different scenarios of driving. Two in the CarMaker environment and one in the Unity environment. The road geometry used in the Unity environment (Kajaman road [24]) was a six-lane highway with a length of roughly 20km. No speed limit existed for this road, but as highways in Sweden often have a speed limit of 100km/h, such a limit seemed fitting. Figure 5.12 shows a birds-eye view of the complete route.

In CarMaker, the Volvo Drive Me route around Gothenburg and an excerpt from Riksväg 160 (Rotviksbro to Ucklum) as seen in Figure 5.13 were used. The Drive Me route consisted of approximately 30km of six to four lane highway with a speed limit ranging from 70-90km/h. The excerpt from Riksväg 160 consisted of 53km of two lane country road with speed limits in the range 50-80km/h. For simulations, a permanent speed of 70km/h was used on Riksväg 160 while the proper speed limits where kept for the Drive Me route. In addition, the VCC Drive Me route was evaluated for day time driving while Riksväg 160 was evaluated for night time driving.

Simulations[5] show that the model keeps the vehicle properly aligned in the lane throughout the complete evaluation runs and manages to drive without issue. Some sub-optimal alignment of the vehicle is visible at times (the vehicle being a bit too close to either lane marking), but considering the environment around it, the behavior seems acceptable. In the case of the Drive Me route for example, safely cutting a corner a bit in a steep curve at 70 km/h seems like an acceptable thing to do since most humans drive in a similar fashion. Based on the measure of performance

[5]https://youtu.be/pT4D1zS6qz4
https://youtu.be/5nvcDgFTOJ8
https://youtu.be/iaR6o8PqJgo

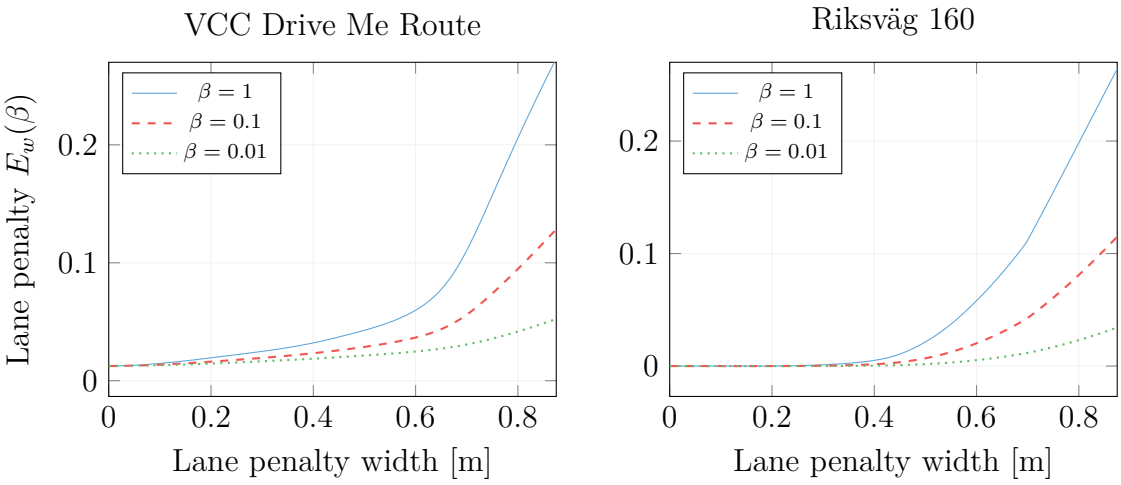


Figure 5.14: Lane penalty metric $E_w(\beta)$ for lane penalty width $w \in [0, 0.875]$ m and $\beta \in \{0.01, 0.1, 1\}$. Vehicle–to–lane distances were collected for the 30km VCC Drive Me route and 53km stretch of Riksväg 160 when the vehicle was controlled by the model trained on Volvo expedition data. The total lane widths are 3.75m and the test vehicle width was 2m.

used in [8], the level of autonomy achieved was either very close or equal to 100% for the performed tests.

During simulation in IPG CarMaker, the lateral acceleration of the vehicle as well as the vehicle–to–lane marking distances were logged for the evaluation runs. In the following sections, the results obtained using these time series are presented based on the penalty metrics defined in Section 5.5. As the proposed lane penalty metric has two hyperparameters, these needed to be tuned for the road conditions and desired driver style. Preferably this would have been done by performing several test-runs with real drivers, logging both (subjectively) good and bad driving behavior. Unfortunately, real vehicle tests were not possible to do in this case, and as such, no tuned values for these parameters are presented. Instead, the performance was determined for a variety of parameter configurations.

5.6.1 Lane Positioning Performance

The lane penalty width (LPW) w_l (left) and w_r (right) were assumed to be equal and constant (not time dependent) for each simulation run. The lane positioning penalties obtained for both road geometries can be seen in Figure 5.14 for a range of the LPW parameter $w \in [0, 0.875]$m and $\beta \in \{0.01, 0.1, 1\}$. The values used for β was based on the motivation presented in Section 5.5.1 and the range of the LPW parameter was determined based on the vehicle and lane configurations used for the simulations. In these cases, the lanes were 3.75m wide and the vehicle had a width of 2m.

The intuition for different LPW parameter values is that an increasing value will penalize deviations from the lane center more. A LPW of zero thus corresponds to no accumulation of penalties, no matter the vehicle's positioning in the lane, as

long as the vehicle remained within it. For the opposite case, a LPW of 0.875m corresponds to penalties being accumulated as soon as the vehicle deviated from the absolute center of the lane.

For these evaluations, the driving performance was deemed (subjectively) good. Therefore, a parameter configuration resulting in a low lane positioning penalty seemed reasonable. A good choice of LPW would then be within the range [0.5, 0.6]m, allowing the vehicle to have a lateral displacement of approximately 0.27 to 0.37m from the lane center without accumulating any positioning penalty. For the different choices of β, this corresponded to the vehicle being well positioned roughly 94–99% of the time while driving on both the VCC Drive Me route and Riksväg 160. The small penalty (1%) even for an LPW of zero for the VCC Drive Me route is a result of the corners being cut in some of the sharp curves of this road.

5.6.2 Level of Discomfort

The lateral acceleration and jerk obtained for a given road geometry provides a measure of the smoothness of the inferred steering actions of the model. For this evaluation, the optimal lateral acceleration was assumed to be known and logged from an optimal trajectory of the road. Preferably, this trajectory would have been obtained by from a human driving the particular route at hand, however in this case, the human behavior was instead approximated with a driver model from the IPG CarMaker software (the IPG driver). Because the IPG driver has full knowledge of the complete simulated environment at all times, its driving behavior could be considered optimal. This IPG driver was set to traverse the simulated route, allowing for the collection of data. This data could then be used to make a comparison of the model performance in terms of the comfort level. The lateral acceleration and jerks were evaluated for both road geometries using the model trained on Volvo expedition data and the IPG Driver. Excerpts of the measured signals can be seen in Figure 5.15, 5.16, 5.17, and 5.18.

From Figure 5.15 and 5.16, one can see that the model generally follows quite closely to the IPG driver in terms of accelerations. For minor road curvatures, it can clearly be seen that the model seems to constantly over and under estimate the required curvature of the road. Nevertheless, for major curvatures, the model seems to perform much better, suggesting that it is better at determining curvatures of actual turns rather than straight sections. The reason behind this more prominent acceleration is based on the fact that the model operates on a "frame–by–frame" basis. For each time instance, the model decides on the appropriate steering command to take, resulting in an oscillatory behavior.

Due to this oscillatory behaviour, there is a significant increase in the jerks measured from the trajectory. From Figure 5.17, and 5.18, one can clearly see a difference between the jerks measured by the use of the IPG driver and that of the model. For minor road curvatures, one can with some effort distinguish that the model to some extent follows the general trend of the curve of the IPG driver, although with much more noise. Similarly as in the case of the accelerations, it is observable that the model seem to perform better for major road curvatures.

From the gathered data, the error metric $E_{\text{lat}}(t; g)$ introduced in Section 5.5.2

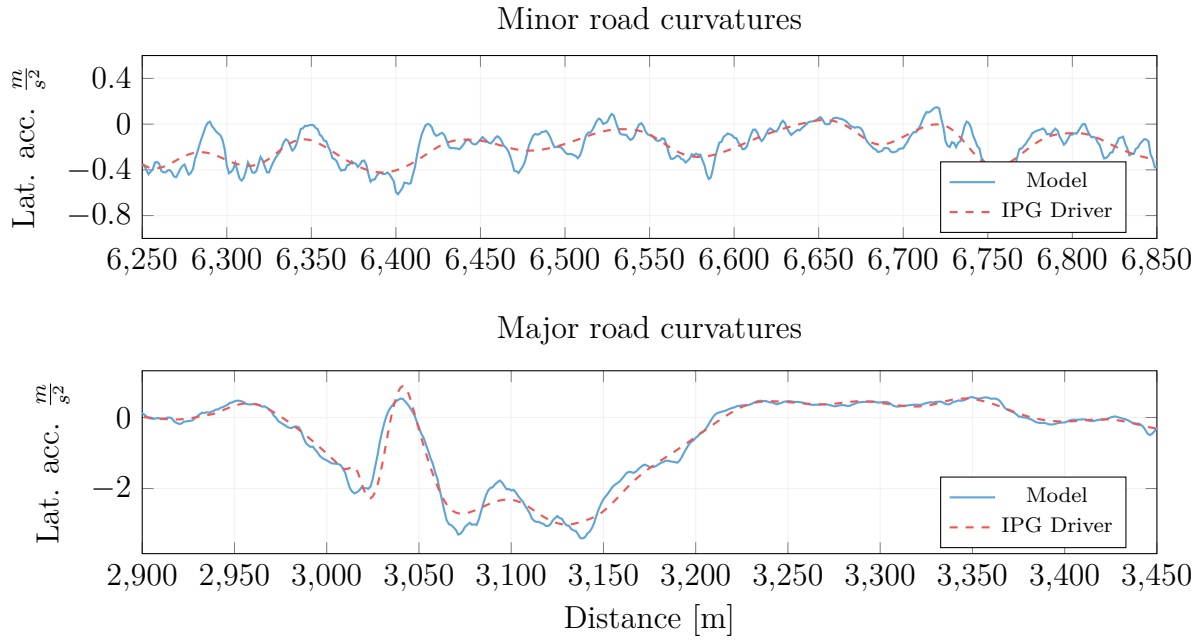


Figure 5.15: Logged lateral accelerations on the Volvo Cars Drive Me route for two different sections of the route with different degrees of curvature. The solid line represents the model trained on the Volvo cars Data and the dashed the default IPG driver.

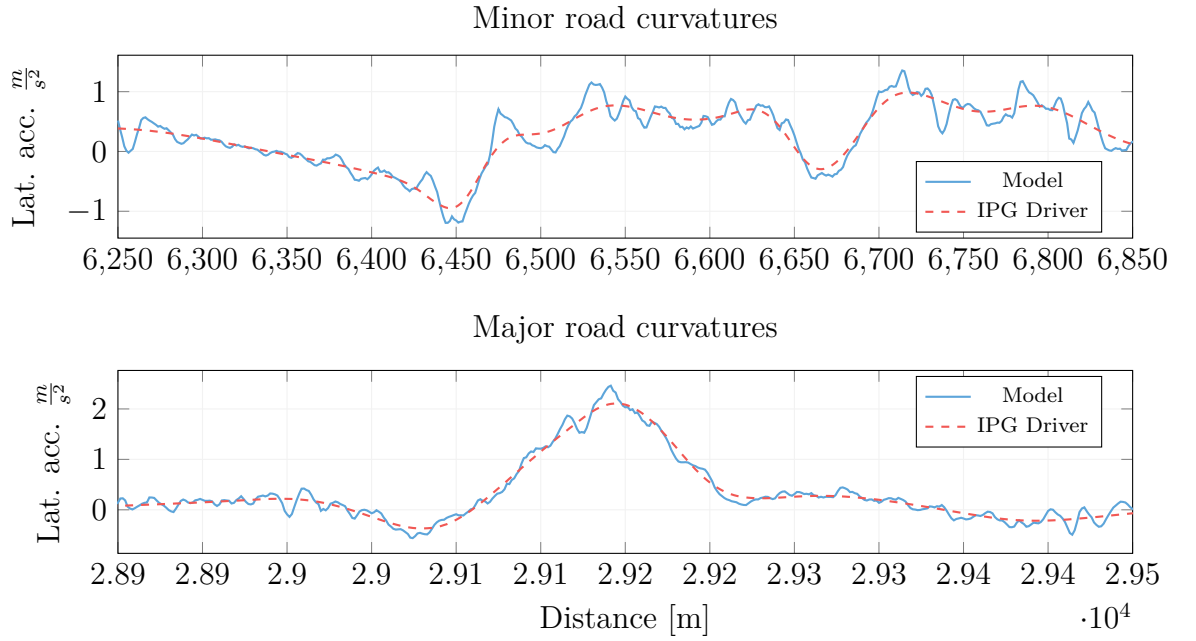


Figure 5.16: Logged lateral accelerations on Riksväg 160 for two different sections of the route with different degrees of curvature. The solid line represents the model trained on the Volvo cars Data and the dashed the default IPG driver.

Table 5.1: Level of discomfort $E_{\text{lat}}(t; g)$ for the model trained on Volvo expedition data and the IPG driver with a comfort threshold $g = 1.8$. The level of discomfort is calculated based on both lateral accelerations and jerks for the Volvo Drive Me route and Riksväg 160.

Driver/signal	VCC Drive Me route	Riksväg 160
Model/acc.	0.1600	0.1112
IPG driver/acc.	0.1404	0.1039
Model/jerk	0.0390	0.0335
IPG driver/jerk	0.0227	0.0070

can be used to compare the level of discomfort of the driving behaviour of the model with that of the optimal driver. A comparison between the model and the IPG driver can be seen in Table 5.1 in terms of accelerations and jerks. From these results, it can be seen that the IPG driver is 1.14 and 1.07 times more comfortable in terms of lateral acceleration for the Volvo Drive Me route and Riksväg 160 respectively. In terms of jerk however, the IPG driver is 1.72 and 4.79 times more comfortable for the same roads.

That the model behaves considerably worse in terms of the level of jerk it not very surprising. Again, one must consider the fact that the model makes decisions in an instantaneous fashion which makes it rapidly oscillate between different steering values. While small, all of the oscillations together add up to form a significantly larger jerk penalty than what an optimal driver would accumulate. So even though the level of discomfort in terms of jerks seems high, it is important to note that the measurements are rather small on average and they might not be perceived as being bad by passengers. Additionally, from the definition of the level of discomfort given in Section 5.5.2, an absolute value of the error of 1 would be at the level at which discomfort ensues. As the model discomfort value is merely a tenth of that, it is well within the comfortable range.

As a final note, in the case of Riksväg 160, the scenario was played out during night with reduced visibility as a result. While this had no effect on the IPG driver (which had knowledge about the simulation state at all times) it made it significantly more difficult for a vision based system to produce proper output, resulting in slightly degraded performance.

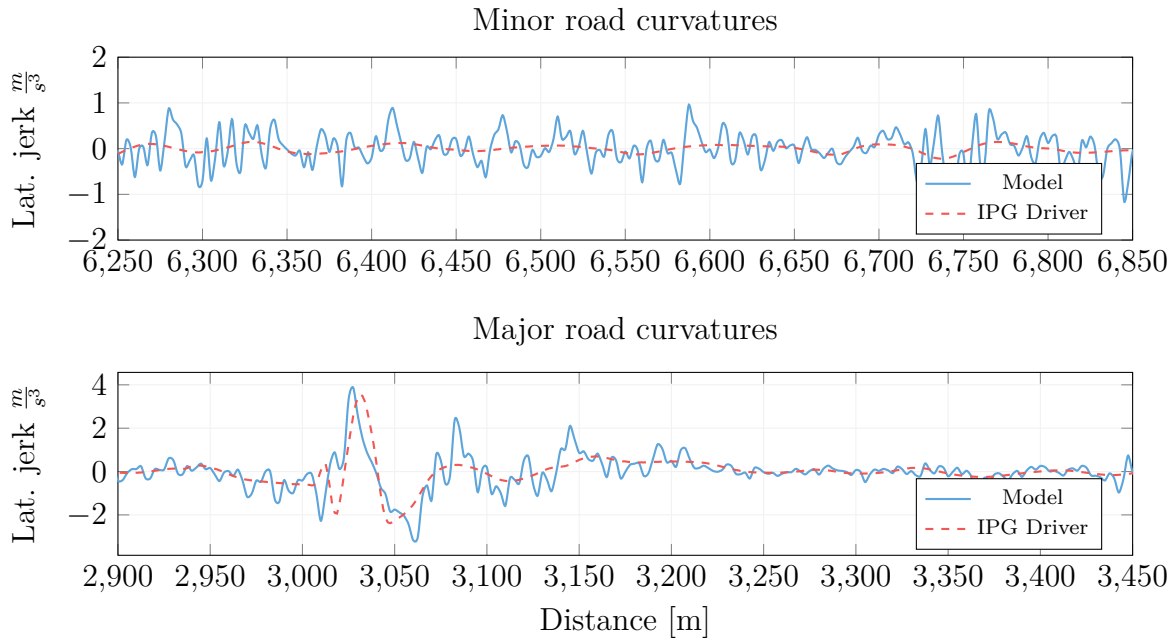


Figure 5.17: Logged lateral jerks on the Volvo Cars Drive Me route for two different sections of the route with different degrees of curvature. The solid line represents the model trained on the Volvo cars Data and the dashed the default IPG driver.

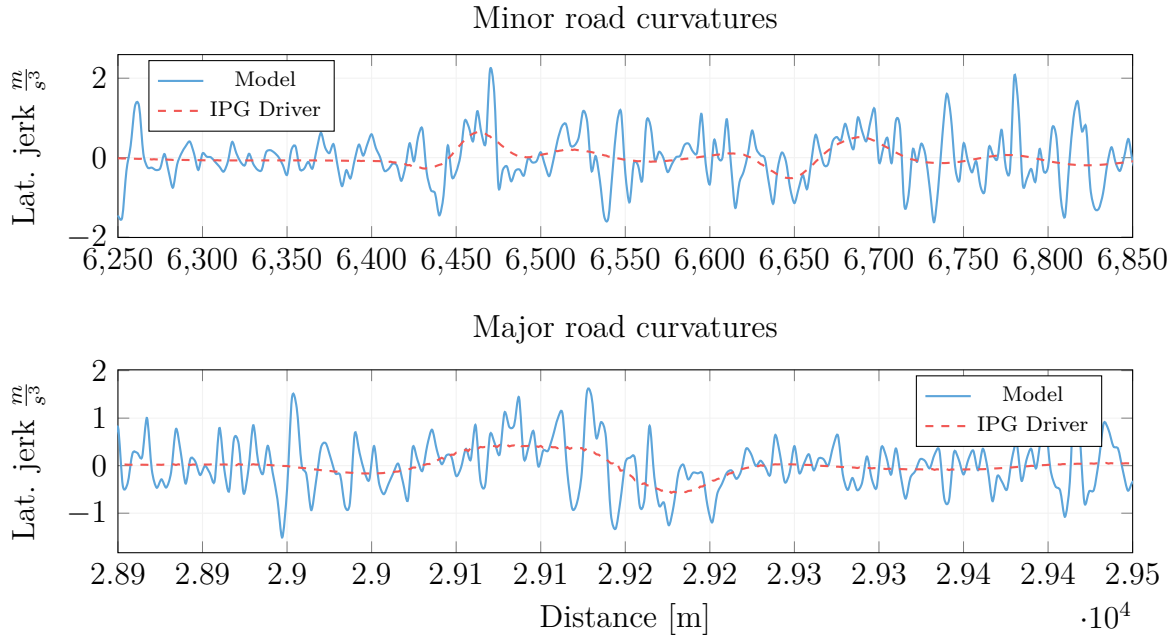


Figure 5.18: Logged lateral jerks on Riksväg 160 for two different sections of the route with different degrees of curvature. The solid line represents the model trained on the Volvo cars Data and the dashed the default IPG driver.

6
Conclusion

With this work, it has been shown that neural network models trained on real world driving data can perform well in simulated environments. That is, they generalize enough to endure a domain change. Notably, the model seems to have learned to react to certain concepts in the real-world data which it then also detected in the simulations as well. Additionally, the detections it makes aligns to some extent with the human intuition of what is important, such as lane markings. This seems to indicate that rapid prototyping, where models are initially tested in simulations, might be a possibility. To assert this however, real vehicle tests are required.

The investigation of the model robustness also revealed that having a sufficiently large data set with enough diversity enables the model to compensate for misalignment on the road. The results show that using a single gray scale front view camera is sufficient in order to attain lane keeping capabilities with a very high level of autonomy. In addition, in simulated environments, the model has also been shown to exhibit robustness to sudden fault injections and subsequent misalignment, seemingly contradicting previously held beliefs.

Overall, the findings in this work seems to indicate that neural networks indeed possess the capacity to act as a holistic solution, at least to certain tasks. With this work, the concept has been verified to work for the basic components of driving, in various environments previously unobserved by the network.

7
Future Work

Over the course of the project, several new ideas and extensions for solving the original task have emerged. The ones deemed most important have been collected and are presented here.

This chapter will begin in Section 7.1 with a discussion on the need to move outside the simulated environments, expressed in the form of performing verification in real vehicles or via hardware in the loop. In the event such real tests proved fruitful, it would have served as good evidence that rapid prototyping indeed had the potential to work. In such a case, further extensions to the simulators would have been interesting as well. In Section 7.2, a brief discussion on the topic of the simulators is provided. The chapter ends with Section 7.3 which elaborates on how the current model architecture might be modified in order to further increase the system's performance.

7.1 Real Vehicle Tests

The results obtained in this thesis seems very promising. At least in the simulated environments the model is able to steer very well. Based on the real data, the general concepts of driving have seemingly been learned. Or at least some sort of minimum representation of them. From this, the conclusion was drawn that knowledge could successfully be transferred across different domains. However, for the results to really have any impact, the resulting model must be evaluated using real tests.

Performing the real vehicle tests would first and foremost answer the question of whether the simulated performance indeed was representative of the one obtained when running the model in the real-world scenarios. In addition, the real-world testing would have been helpful in determining if the sensitivity in camera placement observed in simulations also carried over to reality.

An additional aspect that can be improved via the use of real vehicle tests is that the parametrization of the proposed performance metric can be determined. By collecting data from several real vehicles when driven in a "proper manner" from a human's perspective, the performance metric can be tuned to closely represent good driving. With the metric properly adjusted, newly created models can easily be ranked based on the score they achieve.

Finally, running the software on the target hardware would also have been an indication of how much processing power was consumed and if there was room for more extensions to the model or not. Of course, all of these aspects are naturally very important, and therefore the next logical step would be to test out these initial results in reality.

7.2 Extended Driving Scenarios

In order to verify the model to a greater extent than what has been done so far, more extensive tests are required. For the simulation environments, there are many forms in which such extended tests could come. For example, plenty more road geometries can be included, as the tested routes only amounted to about 100km. More detail might also be added to the already present geometries. Such as landmarks, other vehicles, and weather effects making them more similar to their real counterparts. Additionally, more complex interactions between different vehicles such as merges, overtakes, and right of way need to be modeled and tested as well. There are also many more types of roads to traverse other than highways and country roads. A complete system for autonomous drive would also have to be verified to work for these cases.

One of the main objectives in this thesis was to investigate whether a model trained on real world data would also perform well in simulated environments. That is, the concept of transfer learning was investigated. Because the world portrayed in the simulators in many ways differ from the real world, most notably in terms of graphical fidelity, running models that rely on vision information might be difficult.

However, based on the results obtained, it was shown that the limited graphics nevertheless was enough for the model to drive properly. Although, if the simulator

could produce more realistic images, the issue of transfer learning might be circumvented completely. As of now, we are not yet there, but there are many engines that do provide very realistic graphics. Some of these graphics engines are also working with simulators such as CarMaker in order to take over the rendering of the graphics but leaving the physical interaction and calculations for the original simulator. Using such an additional graphics system would be very interesting for testing out the created model in this work as well as any future models.

7.3 Experimental Model Improvements

In this section, some general ideas to future extensions for the already created model are presented. Most of them were actually implemented, but due to technical difficulties they did not produce any usable output and were therefore not subjected to the evaluation process. In future reiterations of the work, it would be interesting to sort out their issues in order to investigate whether the extensions would improve upon the basic one, or if the general idea behind them is flawed.

First, the idea of introducing the concept of temporal dynamics into the model architecture via several methods is discussed. From there, the section expands briefly on the possibility to include more input into the model as a means to achieve better accuracy. In the end, the concept of changing the problem formulation from a regression task to that of a classification is discussed.

7.3.1 Temporal Dynamics

The performance of the created model experienced some shortcomings as a result of its particular architecture. Most notably, since it operates on a frame by frame basis, it has no notion of what is happening other than that of the current time instant. Because of this, the model has the tendency to rapidly oscillate between different choices for the steering.

Considering that the act of driving intuitively seems to involve a bit more than just instantaneous decisions, a model architecture incorporating temporal information, would probably perform better at the task. When humans are driving a vehicle, it seems reasonable to think that they are not just considering what they see at each and every moment but rather plan ahead. A simple approach for incorporating temporal information, based on the original architecture, was implemented by letting the input consist of a volume of stacked images. This extension required minimal change to the actual architecture, where the only change involved an update of kernel depth for the first convolutional layer in order to accommodate the new input volume size.

Experiments performed using this model did not result in any increase in driving performance. This is most likely due to the fact that the simulation environment interfacing was sub-optimally implemented, which lead to problematic delays in the closed loop evaluation setting, rather than the new model not working. In the future, ensuring that this model is properly tested might yield some insights into how to make the system smoother.

Additional ideas on incorporating temporal information would be to replace and/or extend the baseline architecture with a recurrent model. For example, by including LSTM (Long Short Term Memory) units or GRU (Gated Recurrent Unit) cells. Both of which has proven to be powerful for a large number task requiring the use of sequential data.

7.3.2 Multiple Inputs

The idea of incorporating earlier information (in a simple way) could obviously be extended even further. In addition to passing the n previous images as input, the n last steering actions, as predicted by the model, could also be fed back recursively as additional input. This would allow the model to consider its previous actions, probably increasing the performance of driving.

Other than increasing of the amount of image and/or steering data, additional sensor data might improve on the driving performance even further. During this work, an extension of the baseline model was implemented where the second fully connected layer (50-unit layer) was merged with IMU (Inertial Measurement Unit) information. For this implementation, the vehicle yaw rate, velocity and accelerations were selected as additional inputs. This extension was initially trained on Volvo expedition data but was not subject to thorough evaluation because of the aforementioned technical issues. However, the idea of using IMU measurements as input does, even with the lack of proper result, seem like a good idea given that this type of information most like affect the decisions made by human drivers and is therefore worth keeping in mind for the future.

7.3.3 From Regression to Classification

An alternative approach to the complete problem formulation is to pose the problem of end-to-end inference of steering actions as a classification task rather than a regression task. This would allow for more flexibility in labeling the actions w.r.t. the images. The complete range on possible steering actions could be divided into small discrete ranges, possibly aligned with the distribution of the data. Then, rather than labeling an image with a fixed value for the corresponding steering action, the confidence of the corresponding class could be high, but still allow for some confidence on the closely adjacent classes.

Bibliography

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[2] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649. IEEE, 2012.

[3] Yi Sun, Yuheng Chen, Xiaogang Wang, and Xiaoou Tang. Deep learning face representation by joint identification-verification. In *Advances in Neural Information Processing Systems*, pages 1988–1996, 2014.

[4] Chris Urmson, J Andrew Bagnell, Christopher R Baker, Martial Hebert, Alonzo Kelly, Raj Rajkumar, Paul E Rybski, Sebastian Scherer, Reid Simmons, Sanjiv Singh, et al. Tartan racing: A multi-modal approach to the darpa urban challenge. 2007.

[5] Julius Ziegler, Philipp Bender, Markus Schreiber, Henning Lategahn, Tobias Strauss, Christoph Stiller, Thao Dang, Uwe Franke, Nils Appenrodt, Christoph G Keller, et al. Making bertha drive—an autonomous journey on a historic route. *IEEE Intelligent Transportation Systems Magazine*, 6(2):8–20, 2014.

[6] Dean A Pomerleau. Alvinn, an autonomous land vehicle in a neural network. Technical report, Carnegie Mellon University, Computer Science Department, 1989.

[7] Yann LeCun, Urs Muller, Jan Ben, Eric Cosatto, and Beat Flepp. Off-road obstacle avoidance through end-to-end learning. In *NIPS*, pages 739–746, 2005.

[8] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[9] Andrew Mcafee. What will future jobs look like. `https://www.ted.com/talks/andrew_mcafee_what_will_future_jobs_look_like`, 2013. Accessed 2016-11-22.

[10] Unity game engine. `https://unity3d.com/`. Accessed: 2017-04-12.

[11] IPG Automotive. Carmaker: Virtual testing of automobiles and light-duty vehicles. https://ipg-automotive.com/products-services/simulation-software/carmaker/. Accessed 2017-05-22.

[12] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[13] Michael DeBellis. neuropsychologysketches. http://www.neuropsychologysketches.com/Neurons.html. Accessed 2017-03-04.

[14] Cornell Aeronautical Laboratory Rosenblatt, Frank. The perceptron–a perceiving and recognizing automaton. 1957.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[16] Richard S Sutton. Two problems with backpropagation and other steepest-descent learning procedures for networks. In *Proc. 8th annual conf. cognitive science society*, pages 823–831. Erlbaum, 1986.

[17] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[19] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[20] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[21] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[22] Udacity. https://www.udacity.com/. Accessed: 2017-05-18.

[23] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[24] Kajaman's Roads Free. https://www.assetstore.unity3d.com/en/#!/content/52628. Accessed: 2017-05-18.

[25] Trafikverket. Sektion landsbygd - vägrum 5 vägtyper. http://www.trafikverket.se/TrvSeFiler/Foretag/Bygga_och_underhalla/Vag/Vagutformning/Dokument_vag_och_gatuutformning/Vagar_och_gators_utformning/Sektion_landsbygd-vagrum/05_vagtyper.pdf. Accessed 2017-05-01.

[26] Mariusz Bojarski, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Larry Jackel, Urs Muller, and Karol Zieba. Visualbackprop: visualizing cnns for autonomous driving. *arXiv preprint arXiv:1611.05418*, 2016.

[27] Mariusz Bojarski, Philip Yeres, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Lawrence Jackel, and Urs Muller. Explaining how a deep neural network trained with end-to-end learning steers a car. *arXiv preprint arXiv:1704.07911*, 2017.

[28] Joakim Sörstedt, Lennart Svensson, Fredrik Sandblom, and Lars Hammarstrand. A new vehicle motion model for improved predictions and situation assessment. *IEEE Transactions on Intelligent Transportation Systems*, 12(4):1209–1219, 2011.

[29] Emmanuel Felipe and Francis Navin. Canadian researchers test driver response to horizontal curves. *Road Management & Engineering Journal TranSafety, Inc*, 1, 1998.

[30] Jin Xu, Kui Yang, YiMing Shao, and GongYuan Lu. An experimental study on lateral acceleration of cars in different environments in sichuan, southwest China. *Discrete Dynamics in Nature and Society*, 2015, 2015.