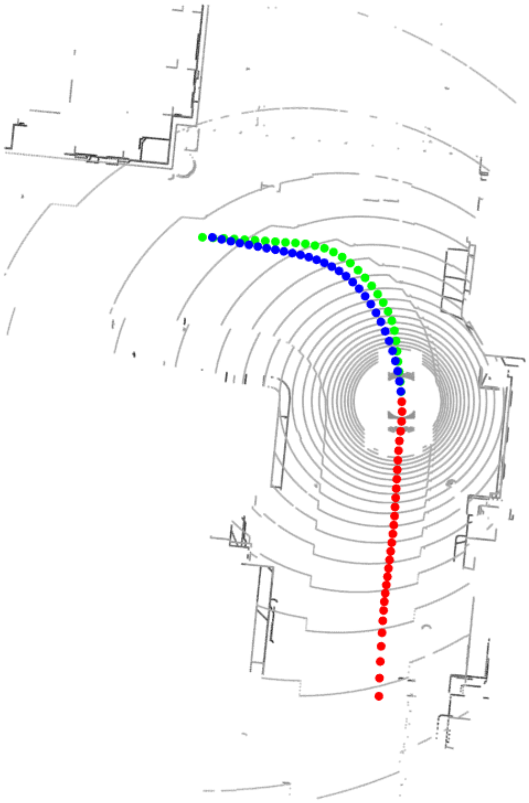
CHALMERS
UNIVERSITY OF TECHNOLOGY



Control of Self-Driving Vehicles Using Deep Learning

Master's thesis in Computer Science – algorithms, languages and logic

JACOB GENANDER
ANNA NYLANDER

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis 2018:EX099/2018

Control of Self-Driving Vehicles Using Deep Learning

JACOB GENANDER
ANNA NYLANDER

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2018

Control of Self-Driving Vehicles Using Deep Learning
JACOB GENANDER
ANNA NYLANDER

Supervisor: Lennart Svensson, Department of Electrical Engineering
Examiner: Tomas McKelvey, Department of Electrical Engineering

Master's Thesis 2018:EX099/2018
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Lidar point cloud seen from above, depicting a simulated car as it takes a left turn. The dotted lines represent the car's past trajectory (red), ground truth future trajectory (green) and predicted future trajectory (blue), as predicted by a deep neural network.

Typeset in LaTeX
Gothenburg, Sweden 2018

Control of Self-Driving Vehicles Using Deep Learning
JACOB GENANDER
ANNA NYLANDER
Department of Electrical Engineering
Chalmers University of Technology

Abstract

This thesis investigates the use of deep learning to control an autonomous vehicle in urban environments. A so called direct perception system was implemented, allowing for human interpretation of the output while providing a means of guaranteed safety. Moreover, the system was trained using unlabeled data, providing a cost and time efficient alternative to other approaches. The data was obtained by recording lidar and GNSS-IMU sensor readings while driving a simulated car. Together with high level directions, traffic light status and speed limit, this information captured the state of the ego-vehicle and its surroundings at every time step. Given such a state, the networks were trained to predict the future trajectory of the ego-vehicle. The predicted trajectory could then be followed by applying appropriate control signals to the vehicle, found using a model predictive controller. Four types of deep neural networks were implemented in order to explore different ways of predicting trajectories. The performance of the networks was measured using the mean squared error between the ground truth trajectories and the predicted ones. In addition, a more qualitative analysis was made by visually inspecting the trajectories. Three of the four network types obtained similar performance, while all networks failed to reach acceptable performance. The networks were able to predict acceptable trajectories in some situations, e.g., turning at crossroads and driving on straight roads. On the other hand, some situations were evidently more difficult, e.g., non-crossroad turns and stopping for traffic lights and other vehicles. Owing to these results, it is not possible to conclude whether the sensors provide sufficient information to control a vehicle or not. In conclusion, deep learning seems to be a promising technique for controlling autonomous vehicles, although the problem was harder than expected.

Keywords: Machine Learning, Deep Learning, Autonomous Driving, Direct Perception, CARLA Simulator, LIDAR, Trajectory Prediction.

Acknowledgements

First and foremost, we would like to thank our supervisor Lennart Svensson for giving us the opportunity to be part of this exciting and interesting project. From the start, he has encouraged our experimentation and ideas and he has also given advice and shared interesting thoughts. We would also like to express our deep gratitude to Juliano Pinto and Dapeng Liu for implementing the MPC, discussing ideas and assisting in general. They have both been great sources of inspiration. Assistance provided by Anders Karlsson was greatly appreciated — we always left his office energized and motivated. Thanks should also be given to Luca Caltagirone for discussing some of the key concepts of the project, to Samuel Scheidegger for helping us with his PyTorch expertise and to Mikael Kågebäck for sharing his knowledge on RNNs. Finally, we wish to thank our examiner Tomas McKelvey for reviewing the thesis, and the E2 department for providing us with a place to work in and an SSD to put the data on.

Anna Nylander, Gothenburg, August 2018
Jacob Genander, Gothenburg, August 2018

Contents

List of Figures

List of Tables

Nomenclature

ANN	Artificial Neural Network
BPTT	Back-Propagation Through Time
CNN	Convolutional Neural Network
FC	Fully Connected
GNSS	Global Navigation Satellite System
GRU	Gated Recurrent Unit
IMU	Inertial Measurement Unit
LIDAR	Light Detection And Ranging
LSTM	Long-Short Term Memory
MPC	Model Predictive Control
MSE	Mean Squared Error
RNN	Recurrent Neural Network
SLAM	Simultaneous Localization and Mapping

1
Introduction

Recently, self-driving cars have become an increasingly hot topic, and have gained a lot of attention from media. Both academia and industry have invested great amounts of resources in the development of fully autonomous cars. Companies such as Waymo[1] and Tesla[2] are already testing their autonomous cars on the road, but none of them have yet reached full autonomy.

By populating the roads with autonomous vehicles, it is argued that numerous areas will see positive effects. Not only might self-driving cars decrease the number of fatal accidents, but it may also lead to less traffic congestion and higher comfort for humans. It is also believed that self-driving cars will utilize the infrastructure more efficiently, and lower the environmental impact. The expected benefits are many, but there is still a long way to go before full autonomy of vehicles becomes reality.

Developing a system which can control a car is a very difficult task. A typical system needs to accurately and quickly scan and interpret a dynamic environment as well as read and understand the intentions of other road users. The system should then calculate the best action to safely and efficiently reach the desired destination, while also communicating its intentions to other road users. The system must also be extremely robust as one small error could lead to fatal accidents.

1.1 Background

The development of fully autonomous cars is not complete, and novel methods to address the problem pop up regularly. However, there are several established approaches and design choices, all of which one has to consider before attempting to create an autonomous vehicle.

1.1.1 Mediated Perception vs Behaviour Reflex

When approaching autonomous driving, methods are commonly divided into two main paradigms; *mediated perception* and *behaviour reflex* [1]. With the mediated perception approach, the system is divided into several sub tasks which is later

[1] https://waymo.com/

[2] https://www.tesla.com/autopilot

melded into one system. A common first step is to divide the system into an environment perception part, an action decision part and a control module. The environment perception part is where sensor data is gathered and interpreted. This part could locate and identify objects. The action decision part is where the next actions are calculated and evaluated. Lastly, the control module is where the final steering signals are derived. Each part can then be evaluated, improved or replaced independently of the rest of the system. An advantage to this approach is that it provides insight into the process and a human can interpret the system step-by-step. This can be very beneficial for debugging, parameter tuning, and also opens up the possibility of integrating constraints on the outputs. The constraints could act as fail-safes in case the system malfunctions.

On the other hand, behaviour reflex models learn a single function for the whole system, leaving the algorithm in a black box which outputs steering signals to the car. In [2] and [3], different methods of environment perception are presented. One advantage of such monolithic systems is that the tedious task of data annotation can be skipped, as the model is trained on basically raw data (which is not the case for mediated perception). This end-to-end approach was utilized in [4], which developed an *artificial neural network* (ANN) which takes video frames as input and outputs the steering angle for an autonomous vehicle. However, no throttle or velocity is predicted in [4]. As mentioned in the same article, end-to-end systems self-optimize according to the task, e.g., to produce steering signals. This means that the network learns the task without solving explicitly defined sub tasks. Sub tasks which are essential for human drivers to master (e.g., identifying the drivable road) may not necessarily be optimal for an autonomous vehicle. A self-optimizing system usually yields a smaller, more effective ANN, compared to a modular approach (i.e., mediated perception) [4].

However, as an alternative to these two extremes, an intermediate approach can be taken, called *direct perception* [1]. The idea with direct perception is to provide insight into the system and enable use of constraints, while also keeping the advantages of a monolithic behaviour reflex system. The authors of [1] demonstrate an ANN which successfully learns to map raw pixel data to a number of parameters, including distance to preceding cars and to lane markings. These values are then used to control a simulated car. The authors of [5] provide another system which produces a cost map over future positions. This cost map is used in a *model predictive controller* (MPC) to calculate steering signals. In 2017, [6], developed a technique for autonomous vehicles using a similar direct perception approach. They attain very promising results, and a demo can be seen online[3].

1.1.2 SLAM vs see-and-drive

Another design choice to consider is whether to use map-based localization and action planning or using a more direct ”see-and-drive” approach. The former is based

[3] Link to demo videos: http://goo.gl/ksRrYA

on the *Simultaneous Localization and Mapping* (SLAM) problem, as described in [7, 8]. A SLAM-algorithm should map the environment as well as simultaneously localize its own position within said map. Furthermore, the algorithm should recognize changes to the environment and update its map accordingly to accurately portray its surroundings. As noted in [7, 8], there are several approaches to solving the SLAM problem, but many practical complications arise when applying it in the real world.

Using GNSS (Global Navigation Satellite System) to locate the vehicle might seem like an obvious solution, but the availability and accuracy of GNSS is a big issue. The signals can be affected by atmospheric conditions and the "urban canyon" problem describes the issue of signal shadow in areas with tall buildings or forests. Moreover, localization errors are hard to detect which could have substantial consequences [7], e.g., driving on the sidewalk or colliding with buildings.

Another common problem with SLAM-algorithms is drift, i.e., small localization errors growing, or drifting, with time. There are numerous methods explained in [7] which reduce the drift of the trajectory, but currently there is no technique which avoids drift entirely. The only way to efficiently counter the drift problem is to ever so often correct the position with a known reference [7]. By using preexisting maps, it is possible to find such a reference by establishing mapping between the input sensor readings and the map.

However, the maps must contain enough information to localize the vehicle regardless of season, weather or traffic situation [7]. Creating such an information dense high-precision map is very expensive and time-consuming. Moreover, the map must be kept updated with the latest information as the roads and environment change, which is not a trivial task.

In contrast to a map-based approach there is a "see and drive" approach. It resonates more directly with how humans drive, and bases decisions and actions on the perceived surroundings. While, of course, the vehicle still needs some kind of directions to reach an end-goal, it is not an essential part of the actual driving and can be outsourced to existing navigation solutions. Advantages of such an approach is that no high precision map needs to be created or maintained and it can drive in an unknown territory. Furthermore, it is not dependent on being online or being connected with other vehicles, therefore it can integrate seamlessly with our current infrastructure.

Even if SLAM seems to be the most used technique today, it is worth investing resources in developing a good independent see-and-drive system. As localization is an extremely important part of current autonomous cars, it is important to make the systems robust and have redundancy. By complementing the system with other techniques which utilize different sensors and information, the system becomes more reliable.

1.2 Objective

The project will focus on the above described see-and-drive approach. More specifically, a direct perception system will be implemented using deep learning and a model predictive controller. The direct perception system should then drive a simulated car according to traffic rules. The neural network will predict a future trajectory, conditioned on the current state of the ego-vehicle and its surroundings. The state consists of measurements from *lidar* and *global navigation satellite system* (GNSS-IMU) sensors, intentions, traffic light status and the current speed limit. The model predictive controller will then take the predicted trajectory as input and output a sequence of timed control signals, i.e., throttle, brake and steer values for each time step in the sequence. When applying the sequence of control signals to the vehicle, the resulting trajectory should align with the predicted one.

The aim of the project is to evaluate how well different neural network architectures can drive an autonomous car, while still keeping the output human-interpretable. The project will use a simulator named CARLA, an open source autonomous driving simulator created by Dosovitskiy et al. in 2017 [9]. As shown in [6], lidar and GNSS-IMU readings together with intentions seem to hold enough information to yield good driving, motivating the choice of sensor data.

Consequently, the objectives are to determine:

1. How well a system with human-interpretable output can drive an autonomous car.
2. Which neural network architecture yields the best result in terms of autonomy.
3. If lidar combined with GNSS-IMU data is enough to drive a car.

1.3 Data

The current state of the vehicle and the world around it must be perceived by the driver (human or mechanical) in order to plan its trajectory. A multitude of sensors are available in CARLA, e.g., cameras, depth-cameras, lidar, GNSS, IMU, etc. Lidar and GNSS-IMU will mainly be used, while no traditional camera feed will be available to the network. Compared to lidar, the spatial information in cameras is limited, due to perspective. Thus it is reasonable to believe lidar is the superior option, as it captures the three dimensional relationship between the sensor and its surroundings.

1.3.1 Lidar

A lidar (light detection and ranging) sensor comprises a number of lasers positioned in a fan pattern. Each laser shoots a beam of light and a sensor records the reflection. The time between shooting and receiving a reflected light pulse is measured, and the distance to the hit object can be calculated. Together with the orientation of the lidar unit, the vertical and the horizontal angle of the beam, the point of reflection can be positioned in 3D space relative to the lidar unit. The resulting collection of 3D points is called a *point cloud* [10].

1.3.2 GNSS-IMU

GNSS is a navigation system based on satellites. The position is based on the distance between multiple satellites and calculated using triangulation. While it is commonly used today, the signals are affected by atmospheric conditions and could fall victim to blockage by tall buildings, thus it is not a reliable system [7].

Inertial Measurement Unit (IMU) is a combination of accelerometers, gyroscopes and sometimes magnetometers. It can measure forces acting upon the object and it's angular rate. It is often used in, e.g., air crafts, vehicles or smart phones.

1.3.3 Intentions

The promising results in [6] were obtained using something they call *intentions*, which provide a sense of direction and planning. An intention consists of *intention direction* and *intention proximity*, i.e., turn **right** in **50** meters. An advantage of using this kind of navigation is its similarity to the input humans use. Hence the intentions can easily be accessed and extracted from existing apps, e.g., Google Maps API. It is shown in [6] that by providing an intention to the ANN, the path prediction improved significantly.

1.4 Scope

As there are many interesting scenarios to investigate when creating an autonomous vehicle, the scope of this project is limited to keep it feasible. Firstly, the performance of the systems is limited by the data available. In real life, an autonomous vehicle would need to handle roundabouts, crossings without traffic lights, parking, etc. However, the traffic situations in this project are those available in the simulator, limited to making simple turns, starting and stopping according to traffic lights, following three different speed limits and adapting to other vehicles and pedestrians. Furthermore, the traffic light status will be assumed to be known even though no camera is utilized. The autopilot provided by the simulator will be used to create the ground truth driving. Although it does follow traffic rules by default, it is not very sophisticated. For example, throttle and brake values are set to one of three states (e.g., no throttle, half-way throttle and full throttle), which might be considered a limitation.

The project will only evaluate the ideas presented in Section 3.3, using the data presented in Section 1.3. The computational speed (hardware listed in Section 4.1) will also be a natural limitation.

1.5 Contribution

This thesis contribute to the field of autonomous driving and specifically to the idea of using deep learning to create a direct perception system. Such a system would allow for human interpretation of its behaviour, while providing a means for guaranteed safety, using constraints and common optimization techniques. Four ideas are investigated and compared in the search for a neural network which can predict future trajectories of a vehicle. We find that none of them reach a satisfactory level of performance, but also that a deep learning direct perception system as a concept has potential. In particular, we find that three of the network ideas have similar performance, while another type of network is significantly inferior.

1.6 Thesis Outline

After this chapter, an extensive theory chapter is presented which explains the underlying concepts and math. It aims to provide the reader with the necessary understanding in order to follow the rest of the thesis. Chapter 3 introduces the overall structure of the system and Section 3.3 proposes the four main ideas. The main ideas refer to the different approaches taken to solve the problem presented above. The following part, Chapter 4, discusses the method. This includes, but is not limited to, a layout of the tools used (Section 4.1), a closer look at the data and how it is processed (Section 4.2 and 4.3), and a detailed description of the neural networks (Section 4.4). The results are presented in Chapter 5 which is followed by a discussion in Chapter 6. Finally a conclusion is given in Chapter 7.

2
Theory

In this chapter we explain the theory behind key concepts used in the project. Sections 2.1 and 2.2 cover the basics of artificial neural networks and their basic structure. Section 2.3 and 2.4 thoroughly disclose the used activation- and loss functions. The two optimization algorithms for neural network training are discussed in Section 2.5.

With a good foundation, a more in depth explanation of the relevant types of neural networks is covered in Section 2.6. After that, Section 2.7 covers the most used regularization techniques for neural networks. Thereafter Section 2.8 discusses the control module which produces the steering signals to the car. Finally Section 2.9 to Section 2.12 covers some key concepts which will be the foundation of the main ideas in the project.

2.1 Machine Learning and Neural Networks

When applying machine learning one often seeks to approximate a function $f(x) = y$, from some example input data x with corresponding output data y. The big advantage of using machine learning is that a model can be used to approximate very complex functions, without having the model's parameters explicitly defined from the start. Instead, the model can *learn* by being exposed to data. The base principle is that the model takes (small) steps towards finding a better approximation to a function, thus better describing the desired output.

Within machine learning there are three main categories: *supervised learning*, *unsupervised learning* and *reinforcement learning*. Depending on the problem at hand, one or the other is preferred. In supervised learning, the model is given some input data x and some corresponding correct answers y, often referred to as the *ground truth*. The model should then find a specific function $f(x) = \hat{y} \approx y$ which maps the input data to the ground truth. For data without a *known* correct answer y the function should hopefully learn to find a good approximation of the correct answer. Unlike supervised learning, the data in unsupervised learning has no explicit mapping from input to output, and thus no ground truth. The main idea behind unsupervised learning is instead to find some function which capture some features of the data. Some examples are finding clusters, detecting anomalies or synthesizing new data. Likewise, reinforcement learning requires no input-output mapping, but instead tries to maximize a reward. This is done by letting an agent, e.g., robot or software, interactively perform actions in an environment, according to some learned action policy that tells the agent how to behave. The policy is learned by ”trial-and-error” exploration, where early actions may affect the reward at a later time. Consequently, the agent must alternate between exploring new policies and exploiting the currently best one, as discovering new action policies is likely to increase the long term reward [11].

While the aforementioned division is based on which environment and data is available, one can also divide the techniques depending on the type of problem, namely: *classification* or *regression*. A classification problem decides which class(es) a type of data belongs to, e.g., which genres does a text belong to. In contrast to the discreet classification, regression is used to estimate continuous values, e.g., the expected weight of a newborn baby. Whether to use classification or regression depends on what problem we want to solve.

As implied before, machine learning is centered around data. The data is usually split into three data sets with different purposes. Firstly, the *training set* is the data the algorithm trains on. This set should be the largest and cover all types of data the algorithm is expected to handle. Secondly there is often a *validation set*. The algorithm should not train (update parameters) on the validation set, rather it is used during training as an assurance that the algorithm generalizes well to unseen data. *Hyperparameters* are often tuned manually to obtain good results on the validation set. Finally the *test set* is used only when the algorithm has finished training. The purpose of the test set is to evaluate the final model on data which has not been used in any part of the training, as an unbiased measure of its performance.

This project is mainly focused around supervised learning. Both classification and regression is explored in different ideas, but in all ideas the corner stone is *deep learning*. Deep learning is a niche of machine learning where one uses *deep neural networks*. Before looking at deep neural networks, the basics need to be understood.

2.2 Feed-Forward Neural Networks

The simplest type of ANN is a *Feed-Forward Neural Network* (FFNN). It was first devised in the 1940s, gathering inspiration from the human brain [12]. Since then many improved approaches and techniques have been devised, but the foundation is still the same.

An FFNN consists of several *neurons* which can be seen in Figure 2.1. Each neuron j in the hidden layer gets some input values $x_1, x_2, ..., x_k$ which are multiplied with its respective synaptic weights $w_{j1}, w_{j2}, ..., w_{jk}$. The products are then summed together with a bias b_j. The value of the summation is then passed trough an *activation function* $\sigma(\cdot)$, which results in the output o_j. The complete equation for

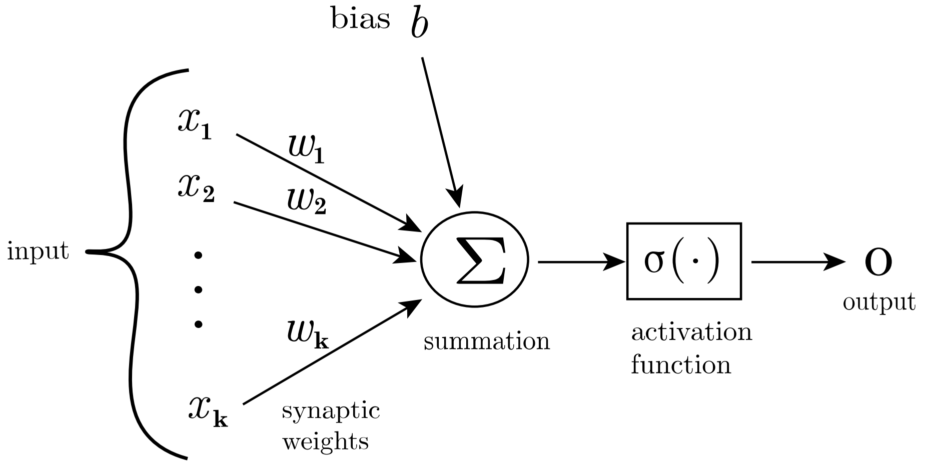


Figure 2.1: The typical structure of a neuron used in neural networks. The output of a neuron in general is here denoted o.

the output of neuron j is thus

$$o_j = \sigma\left(\sum_i w_{ji}x_{ji} + b_j\right). \tag{2.1}$$

The output of neuron i in the following layer is computed using the same procedure, but with the output of the previous layer as its input.

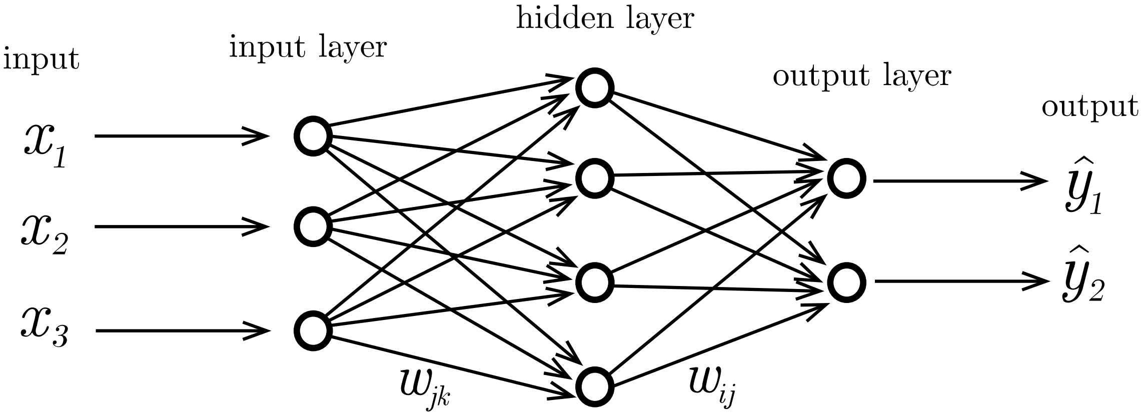


Figure 2.2: An FFNN with an input layer, one hidden layer and an output layer. Each circle represents one neuron, as depicted in Figure 2.1. The output of the final layer corresponds to the approximation $\hat{\boldsymbol{y}}$ of the ground truth $\boldsymbol{y}$.

The neurons of an FFNN may be connected in several layers (called perceptrons). The neurons are connected using the aforementioned weights in an acyclic fashion, where the output of the neurons in one layer is the input of the neurons in the next layer. Passing data through a network will be denoted as $f(\boldsymbol{x}, \boldsymbol{\theta}) = \hat{\boldsymbol{y}}$ where $\mathbf{x}$ and $\hat{\boldsymbol{y}}$ is the input- and output data respectively, and $\boldsymbol{\theta}$ represents the parameters of the network, i.e., weights and biases. Figure 2.2 depicts an example of such a network. This neural network in particular is *fully connected*, which means that all neurons in one layer is connected to all neurons in the following layer.

The size of the input layer is determined by the shape of the input data, e.g., if the input data is an image, then each input x may represent one pixel. The size of the output layer is dependent on the shape and size of the desired output. Depending on the problem, the activation function might differ between the layers, but usually the same activation function is used throughout the network. An exception might be the last layer, which needs to produce values in the range of the target values $\boldsymbol{y}$. The most important purpose of the activation function is to introduce non-linearity into the otherwise linear model.

Naturally, the architecture of the network has a huge impact on performance. For example, the number of layers, size of layers, and hyperparameters can all affect the network greatly. On non-trivial data sets, i.e., the data having relationships which are hard to approximate, it is quite difficult creating an effective network from scratch, which can model these relationships. Furthermore there is no easy way to tell beforehand whether an architecture will perform well or not. Currently, the only established way of evaluating an architecture is simply to train the network and measure its performance — a potentially time consuming process.

2.2.1 Training a Network

Training of a supervised neural network, as described in [13], usually starts by initializing all parameters to small random values. The parameters are then adjusted in order to minimize the distance between the output of the network and the ground truth, thus being an optimization problem. This distance can be calculated using a suitable function for the problem at hand, e.g., the L2-norm, which measures the squared distance between the output and ground truth vectors. The distance is then used to construct a scalar valued *loss* or *error*, denoted by L, which can be optimized. Finding a good *loss function* $L = L(f(\mathbf{x};\boldsymbol{\theta}),\mathbf{y})$ is important in order for the network to learn correctly.

In order to adjust the parameters of neural networks one common method is *gradient descent*, also known as steepest descent. In every step of the training, the gradient of the loss w.r.t. the parameters, $\nabla_{\boldsymbol{\theta}}L(\cdot)$, points in the direction of the steepest increase of loss. Hence, subtracting the gradient from the parameters moves their values towards producing a lower loss value.

The learning rate η is set to scale the update, as to not take too large or small steps in parameter space. Too large learning rate value may cause the loss to diverge, while a too small value may cause a too slow convergence [13]. By calculating the gradient of the loss w.r.t. all the parameters, these can be iteratively updated according to

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}}L(f(\mathbf{x};\boldsymbol{\theta}),\mathbf{y}). \tag{2.2}$$

In order to compute the gradient of the loss w.r.t. a specific parameter in the network, the gradients of all parameters between the last network layer and the parameter of interest have to be calculated, as they depend on each other. This is done using the chain-rule of calculus and results in the error being propagated backwards through the network. This procedure is called *backpropagation*.

During training of a supervised neural network, this process is repeated many times over the training data set; first passing a data point forward in the network, then doing backpropagation and updating the parameters. In each iteration, the network becomes a little bit better at mapping the input to the desired output data, and hopefully converges to a satisfying performance.

2.3 Activation Function

The activation function in general and its purpose was briefly discussed before in Section 2.2. In this project, three main activation functions are used: *Exponential Linear Unit* (ELU), *Logistic Sigmoid* and *Softmax*. The first is an activation function commonly used for regression problems, defined as

$$\mathrm{ELU}(x) = \begin{cases} x, & ifx > 0 \\ \alpha(\exp(x) - 1), & ifx \leq 0 \end{cases}, \tag{2.3}$$

where α controls the slope rate where $x \leq 0$. It is similar to the *Rectified Linear Unit* (ReLU), but has negative values which reduce bias of the activation mean and thus speeds up learning [14]. The functions are plotted for comparison in Figure 2.3.

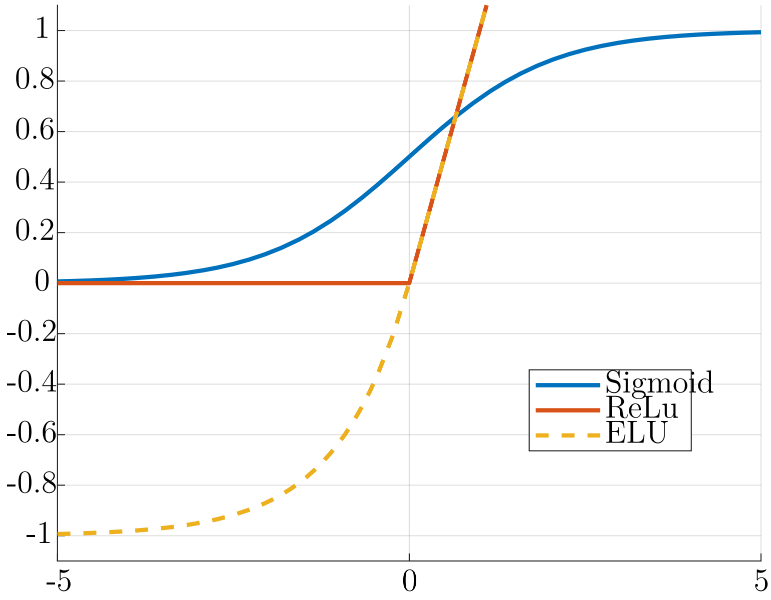


Figure 2.3: The activation functions used. Note that, in contrast to ReLu, ELU can produce negative values.

The logistic sigmoid and softmax functions are quite similar and both are primarily used for classification problems. Both functions convert arbitrary values to a (discrete) probability distribution over class membership and saturate when exposed to large positive or negative values. The main difference between the two functions is that the logistic sigmoid is used when performing binary logistic regression (e.g., hotdog or not hotdog?), while the softmax function is used for multi-class logistic regression. The sigmoid function is defined as

$$\sigma(x) = \frac{1}{1+e^{-x}}, \tag{2.4}$$

converting an arbitrary value x to a value between 0 and 1. The softmax activation function is defined as

$$\sigma(\boldsymbol{x})_j = \frac{e^{x_j}}{\sum_{n=1}^{N} e^{x_n}} \tag{2.5}$$

and scales an N-dimensional vector $\boldsymbol{x}$ of arbitrary values into having all values $\sigma(x) \in (0,1)$ and $\sum_{i=1}^{N} x_i = 1$. Consequently, it can be used to represent a probability distribution, where dimension n of the vector represents the probability of belonging to class n.

2.4 Loss Function

The loss function will differ depending on the network and if the problem is within regression or classification. A common loss function for regression is the *Mean Squared Error* (MSE) described below. It is also the function used in this project to measure the performance of the neural networks. Even in the cases another loss function is used to train the neural network, the MSE is also calculated in order to accurately compare performance.

The MSE is used to compare the ground truth with the prediction. The output of the network is an ordered sequence $\hat{p} = (\hat{p}_1, \ldots, \hat{p}_{n_f})$, consisting of horizontal-vertical coordinate pairs $\hat{p_n} = (x, y)$ relative to the vehicle's current position, predicting its future path. The coordinate pair at index n in the series correspond to the position at time $n \cdot \Delta_t$ seconds from the current time, where $\Delta_t = 0.1$ seconds. The loss L for a predicted trajectory $\hat{p}$ and ground truth trajectory $\dot{p}$ is defined as

$$L = \frac{1}{k} \sum_{n=1}^{k} (\hat{p}_n - \dot{p}_n)^2, \tag{2.6}$$

where the squaring is performed element-wise. The errors in a sequence are here weighted equally.

While MSE is used as the loss function for the regression neural networks, the most commonly used loss function for classification is *cross entropy.* Cross entropy measures the difference between two distributions, i.e., how the output classification probability differs to the ground truth distribution (however, the ground truth is usually only one class, i.e., a degenerate or uniform distribution). The cross entropy between two distributions is defined as

$$H(P, Q) = H(P) + D_{KL}(P||Q), \tag{2.7}$$

where P is the "true" distribution and Q is the guessed distribution [13]. To put cross entropy in context, consider the definition of entropy $H(P)$ of a distribution P:

$$H(P) = -\sum_{i=1}^{n} Pr(p_i) \log\left(Pr(p_i)\right). \tag{2.8}$$

As described in [13], entropy measures how unpredictable the events of the distribution are, i.e., the average amount of information attained from one event drawn from the distribution. The Kullback–Leibler divergence $D_{KL}(P||Q)$ describes the *relative entropy* of P with respect to Q. The guessed distribution Q makes some assumptions to try to look as similar to P as possible, and the cross entropy describes how wrong the assumptions are.

Lastly, a special case of cross entropy is *Binary Cross Entropy*, where only two classes exist. For example, when predicting road regions in traffic images, the two classes could be *is road* or *is not road*. The function is defined as

$$l_n(\hat{y}, y) = -\left(y_i \log\left(\hat{y}_i\right) + \left(1 - y_i\right) \log\left(1 - \hat{y}_i\right)\right), \tag{2.9}$$

in which $\hat{y}$ is the output from the network and y is the target (i.e., ground truth). Binary cross entropy is used as a loss function in the neural networks based on *semantic segmentation*. A theory section about semantic segmentation can be found in Section 2.11.

2.5 Optimization Methods for Gradient Descent

Since training of a neural network can be very time consuming, it is crucial to do it efficiently. In this section the two main optimization methods used in the project will be presented. Both are based on the previously explained gradient descent, but apply some additional heuristics to the optimization process.

2.5.1 Minibatch SGD Optimization

The most naïve way to apply gradient descent is to simply update network parameters using the gradients of the average loss on the whole training set, also known as batch gradient descent [15]. Although the negative gradient may point in a very precise direction of steepest descent, it may take prohibitively long time to calculate gradients on the whole data set. On the other extreme, calculating the gradients using only one input at a time, referred to as *stochastic gradient descent*, may yield too large jumps in parameter space to learn the task at hand efficiently [15]. The stochasticity is a result of calculating the gradient using a random data point instead of using the average of all points.

Consequently, it might be a good idea to calculate gradients using a small subset of μ datapoints, referred to as a *minibatch*. By using randomly sampled minibatches in each iteration of the training, an estimate of the true gradient (i.e., as computed on the whole training set) is made. This way of training, called *minibatch stochastic gradient descent* [15], may speed up convergence and enable training on otherwise prohibitively large datasets [13].

The minibatch SGD optimizer used in this project also utilizes a technique called *Momentum*. Similarly to when using only one data point to calculate gradients, estimates may become very noisy when using a small mini-batch size. Because of this noise, the gradients may vary greatly compared with the gradients computed on the full train set. Therefore, it can be useful to preserve some of the velocity attained by following previous gradients, where the velocity refers to the direction and magnitude of previous gradients. Momentum is a technique which introduces such a velocity parameter, m, which accumulates past gradients while decaying their contributions exponentially each iteration. An illustration of the impact of using momentum is found in Figure 2.4. The velocity update is defined by

$$m \leftarrow \alpha m - \eta \nabla_{\boldsymbol{\theta}} \left(\frac{1}{\mu} \sum_{i=1}^{\mu} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right), \tag{2.10}$$

where the gradient of the average loss in a minibatch is calculated. The learning rate is denoted by η, while $\alpha \in [0, 1)$ defines the rate of decay [13]. The parameters are updated accordingly, using the formula

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + m. \tag{2.11}$$

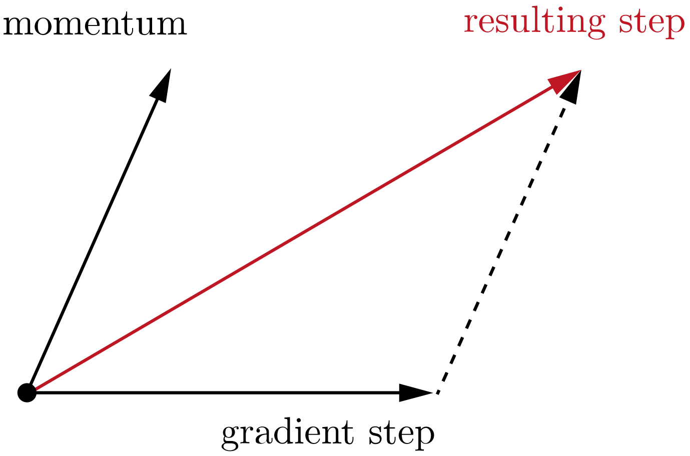


Figure 2.4: The retained momentum is used to find a better step direction. This is especially useful in cases where the gradient direction fluctuates heavily.

2.5.2 Adam Optimization

Finding a good value for the learning rate is of great importance, as it dictates training results to a high degree [13]. There exists methods for finding a good value for the learning rate, as well as ways of applying scheduled modifications to it while training. Nevertheless, approaches to adapting the learning rate to individual parameters have shown to be preferable due to their fast convergence times and great results. In [15], a comparison between some of the most common optimizers is made, suggesting that adaptive learning rate capabilities are preferable when dealing with deep and complex networks. In particular, the *Adam optimization algorithm* is pointed out to possibly be the best in general.

The intuition behind adapting the learning rate to individual parameters is that parameters with a history of small magnitude gradients should be updated with more impact than those with larger ones [15]. Ultimately, training will progress more evenly along all parameter axes [13].

Adam [16] (which is an acronym for adaptive moment estimation) makes use of past gradients, much like momentum. In addition, it also incorporates the square of past gradients. Using the following notation for the gradient g of the loss L w.r.t. parameters $\boldsymbol{\theta}$,

$$g = \nabla_{\boldsymbol{\theta}} L\left(f(x); \boldsymbol{\theta}), y\right), \tag{2.12}$$

the mean m and uncentered variance v estimates are calculated, respectively, as

$$m \leftarrow \beta_1 m + (1 - \beta_1) g \tag{2.13}$$

and

$$v \leftarrow \beta_2 v + (1 - \beta_2) g^2, \tag{2.14}$$

where β_1 and β_2 are decay rates in the interval $[0, 1)$ and g^2 denotes element-wise squaring of g [13]. The estimates are then bias-corrected using the terms

$$\hat{m} = \frac{m}{1 - \beta_1} \tag{2.15}$$

and

$$\hat{v} = \frac{v}{1 - \beta_2} \tag{2.16}$$

Finally, the update rule used in Adam is

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\eta}{\sqrt{\hat{v}} + \epsilon} \hat{m} \tag{2.17}$$

where ϵ is a small valued constant used for numerical stability.

2.6 Networks

In Section 2.2 the basics of neural networks was explained. However there are many variants of networks which are advantageous in different settings. In this projects two main variants where used, namely *Convolutional Neural Network* (CNN) and *Recurrent Neural Network* (RNN). CNNs have seen great success among computer vision tasks and are a natural choice when dealing with spatial data in image format, e.g., the top view of a car's surroundings. Recurrent networks are particularly suited for processing series of data, as they can keep and update an internal state, acting as a memory.

2.6.1 Convolutional Neural Networks

Convolution is an operation which may be applied to N-dimensional operands. For the purpose of simplicity and relevance to this project, the following paragraphs will discuss the two-dimensional case only.

A CNN consists of a number of convolutional layers. Each such layer makes use of a number of convolutional *filters*, or *kernels*. The values of these kernels are learned during training of the network and act as feature detectors. One kernel might for example learn to identify the presence of horizontal edges in an image [17]. Note that the size of a kernel is set during the initialization, and cannot be changed later on. Figure 2.5 depicts one kernel sliding across the image to extract one feature. The resulting array is called a *feature map*. Usually, more than one kernel is used which results in several feature maps. The input image can be seen as a special case of a feature map.

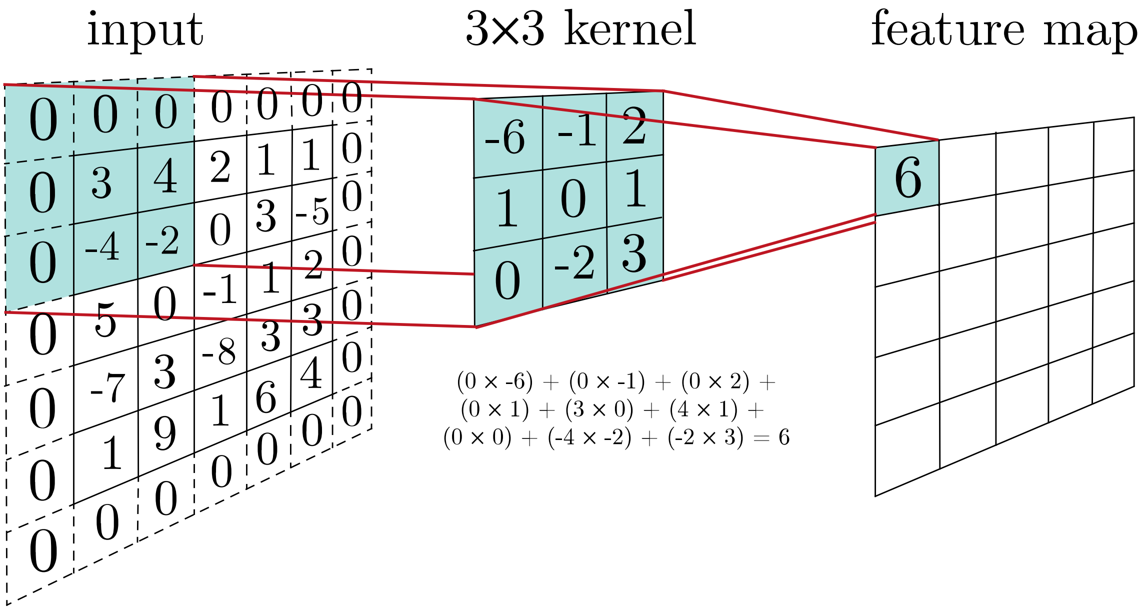


Figure 2.5: A 3x3 kernel is sliding over the input and produces the feature map. The input is padded with zeros in order to maintain the size of the feature map.

The convolution operation consists of sliding the kernel over the image and performing element-wise multiplication with the intensities of the pixels covered by the kernel, producing one value per location. These outputs collectively form a feature map, indicating for all regions the presence of the particular feature learned by the kernel. The convolution operation is often denoted by $*$ and is here defined on a 2D-image I with a kernel K as

$$(I * K)(i, j) = \sum_m \sum_n I(i - m, j - n) K(m, n), \tag{2.18}$$

where i, j are the pixel indices on which the convolution is applied, while m and n are the width and height of the kernel [13]. Note that an individual element in row

a and column b of the matrices I and K is denoted using parentheses rather than subscripts for clarity.

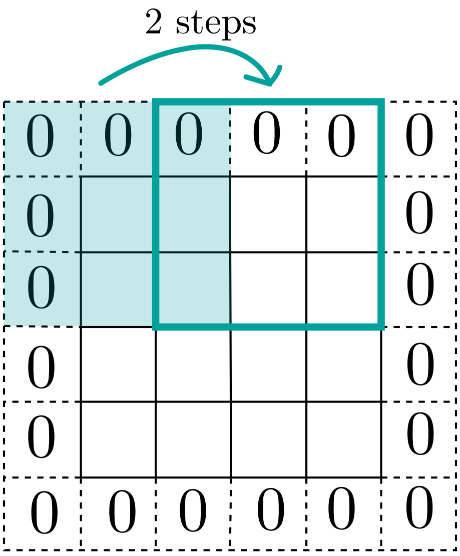
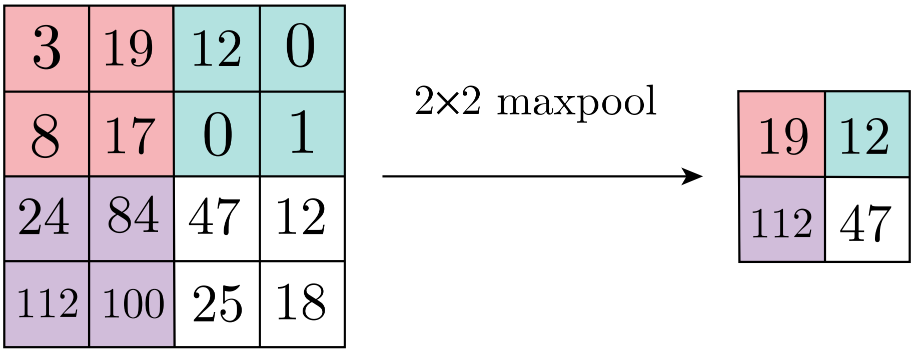


Figure 2.6: One 3x3 kernel moves over an input using a stride of 2. By using a larger stride the resulting featuremap reduces in size.

Convolution with a kernel on an image can be applied with different stride lengths s, meaning that the kernel is moved s pixels horizontally and vertically before performing the operation. Figure 2.6 shows a kernel using $s = 2$.
When doing convolutions the resulting feature map of a convolution will be smaller than the input (except for kernel size 1x1 with stride 1). If one wishes to maintain the input size from layer to layer, padding can be applied. Most often *zero-padding* is used, as zero corresponds to no activation. Note that if the filter size, height or width, does not break even with the number of pixels in the corresponding dimension, the last pixels at the end of the image will be lost.



3	19	12	0
8	17	0	1
24	84	47	12
112	100	25	18

19	12
112	47

Figure 2.7: A small example of max pooling. The input is divided in 2x2 blocks where only the largest value is kept. While max pooling is the most common pooling, there are also min pooling and average pooling.

On the other hand, reducing the size of the input also reduces the demands on computation and memory. This can be achieved by *pooling* neighbouring pixels and reducing them to only one value, e.g., by taking the largest of the values, as is done in *max pooling*, see fig 2.7. Moreover, by summarizing the inputs in this way, the network becomes more robust to small features translations [13].

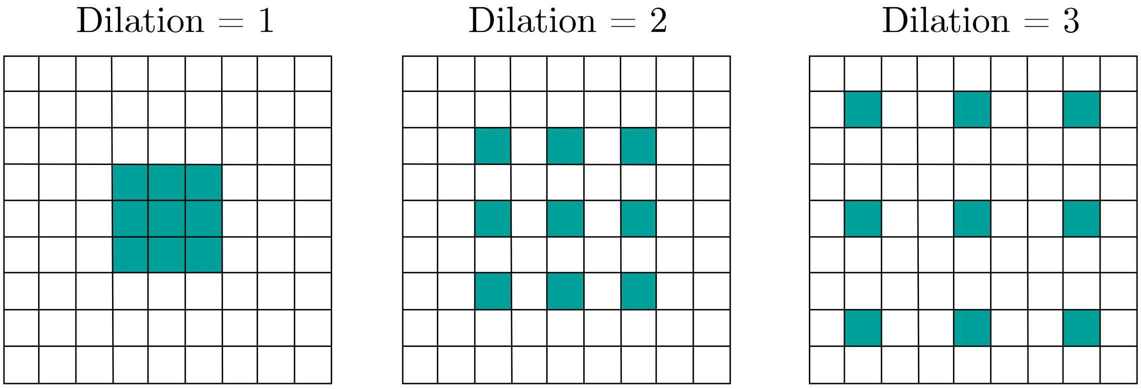


Figure 2.8: A 3x3 kernel with different dilation.

Another possible modification to the kernel is *dilation*. Consider a network with convolutional layers, having kernels of size 3x3 and applying stride $s = 1$. Every unit in the resulting feature map of the first layer then has access to a 3x3 region of the input, i.e., a receptive field of size 3x3. By dilating the elements of kernels in the next layer, each unit in the resulting feature map gains access to information about to non-overlapping regions in the input. It is thus possible to increase the receptive field exponentially with each layer (instead of linearly), while maintaining the number of parameters in them [18]. In fig 2.8 there are some examples visualizing how the dilation affects the kernel.

As each output unit in a convolution layer may be connected only to a small region of the input image, i.e., the kernel is smaller than the image, such networks are said to have sparse interactions [13]. As the parameters of a kernel are shared across every small region in the whole image, the memory requirements can be significantly reduced compared to using a fully connected layer, which could possibly have several orders of magnitude more parameters. As a kernel is applied to all pixels in an image, the corresponding feature may be detected at different locations in the input image and feature maps. This is beneficial, since the same kernel can be used regardless of the feature's location, as opposed to learning location-specific feature detectors [13].

2.6.2 Recurrent Neural Networks

Recurrent neural networks are particularly suited for processing series of data, while allowing for input and output sequence of varying length. RNNs are comprised by recurrent *units*, or *cells*. These terms are equivalent and used to describe smaller networks within the larger network, encapsulating a more complex structure than a single layer of neurons. By updating and transferring a *hidden state* between time steps within a recurrent unit, recurrent networks effectively exhibit memory [19]. By unrolling the time steps, the recurrence can be seen as multiple copies of the same network, connected along the time axis, visualized in Figure 2.9. The memory might be updated using so called *gated units* (e.g., the units explained in Sections 2.6.3 and 2.6.4), but is not a necessity. However, as demonstrated in [19], recurrent units with gates can be superior to simpler units without gates.

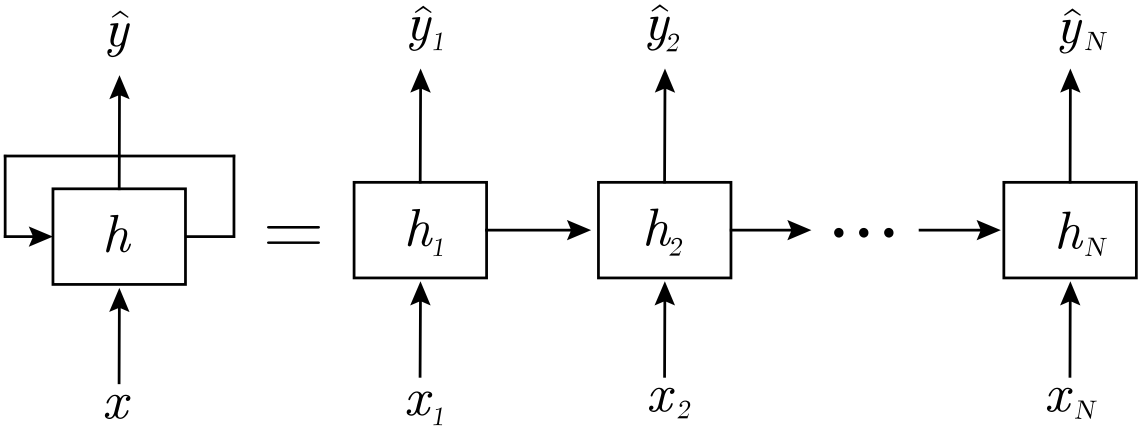


Figure 2.9: To the left is a RNN with input x, output y and a hidden state h. The right hand side shows the unrolled equivalent.

When using simple units without gates for updating the hidden state between time steps of an RNN, the gradients tend to vanish (or explode) as the error is backpropagated. In the case of using, e.g., the hyperbolic tangent function as the activation function, the gradients vanish due to the fact that its partial derivative is always less than 1, except in 0. When gradients are multiplied during backpropagation, the parameter updates become exponentially smaller (or larger with too large parameter initialization values) towards the beginning of the network. As a consequence, learning long-term dependencies in the input may take impractically long time [19] or result in numeric overflows.

The vanishing gradient problem can be alleviated using gated units. The gates operate on the unit's hidden state and input to decide what to remember from previous time steps and how to integrate information from the current input into the memory. The activation function applied on the modified state is the identity, i.e., there is no squashing of the output, meaning that gradients do not vanish nor explode [20].

2.6.3 Long Short-Term Memory

One of the popular gated units is the *Long Short-Term Memory* (LSTM) unit [20] which makes use of three gates, namely the *input gate*, *forget gate* and *output gate*. The LSTM unit operates on the input x, a hidden state h and a cell state c to transfer relevant information between time steps, see Figure 2.10.

The purpose of the forget gate is to regulate the flow of information from the previous hidden state, i.e., what to remember [19]. The activation of the forget gate at time step t, i.e., step t in the input sequence, is defined by

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f), \tag{2.19}$$

where $[h_{t-1}, x_t]$ denotes concatenation of the previous hidden state and the current input. Parameters W_f and b_f are the weights and biases of the forget gate (hence the f in the subscript), while $\cdot$ denotes matrix multiplication. The activation is a measure of inclusion to the next time step.

In a similar way, the input gate is responsible for including the relevant parts of the input and discarding the rest, according to the equation

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i), \tag{2.20}$$

where W_i and b_i are the weights and biases of input gate. A new candidate cell state $\tilde{c}_t$ is formed by

$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c), \tag{2.21}$$

using the same notation principle as before for weights and biases.

The candidate cell state could potentially become the new, updated cell state (hence being referred to as a *candidate* cell state). The actual new cell state c_t can now be formed, using a mix of the old cell state c_{t-1} and the candidate cell state $\tilde{c}_t$. Again, what should be included from respective state is decided by the forget gate f_t and input gate i_t which are used to scale old and candidate cell states. The updated cell state is calculated as

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \tag{2.22}$$

where $\odot$ denotes element-wise multiplication.

The output gate determines what part of the updated cell state c_t are relevant for the output. A vector $\tilde{o}_t$ describing the relevance is formed by

$$\tilde{o}_t = \tanh(W_o \cdot [h_{t-1}, x_t] + b_o). \tag{2.23}$$

Finally, the actual output h_t (i.e., the updated hidden state) of the LSTM unit at time step t is defined by

$$h_t = o_t \odot \tanh(c_t), \tag{2.24}$$

which is the updated cell state activation, scaled by the output gate.
By using the reversed input sequence in addition to the original sequence, [21] show that such a *bidirectional* network can produce better results than a standard unidirectional RNN. The intuition behind using the reversed sequence is that the prediction performed in the current time step can benefit from information available from future time steps as well [21]. This has shown to be useful in, e.g., neural machine translation [22].

2.6.4 Gated Recurrent Unit

Another unit related to the LSTM is the *Gated Recurrent Unit* (GRU). It makes use of a *reset gate* and an *update gate*, while having only one state, see Figure 2.10. The information flowing from the previous state h_{t-1} is restricted by the reset gate, for which the activation is calculated as

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r). \tag{2.25}$$

As the name suggests, the reset gate has the power of resetting the current state, i.e., forgetting previous time steps, either partially or completely.

Likewise, the update gate has the power to decide how much to update the previous state with information from the current input, according to

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z). \tag{2.26}$$

Like in the LSTM unit, a candidate state $\tilde{h}_t$ is computed using the formula

$$\tilde{h}_t = \tanh(W_h \cdot [h_{t-1}, x_t] + b_h). \tag{2.27}$$

The output of the GRU is defined as

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t, \tag{2.28}$$

i.e., as a linear interpolation between the previous state h_{t-1} and the candidate state $\tilde{h}_t$.

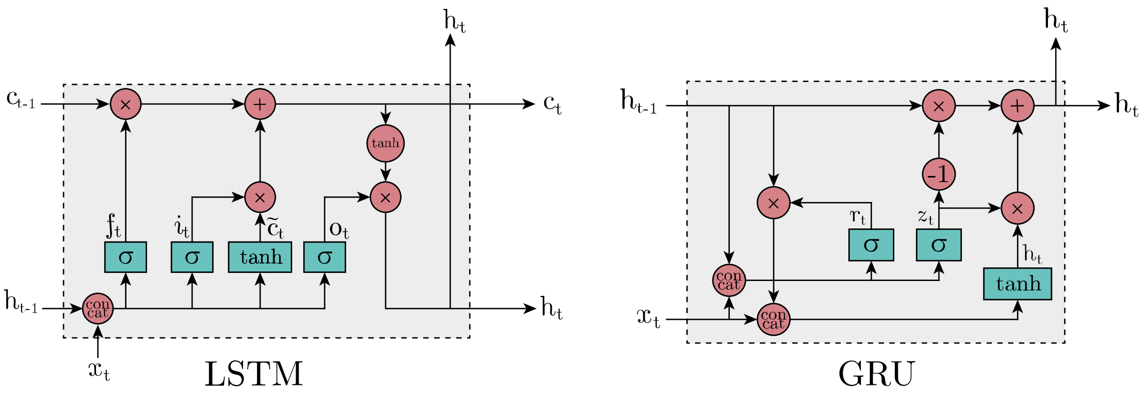


Figure 2.10: Left: LSTM unit where the input x_t and previous cell state c_{t-1} are modified to form the output h_t. The output is in turn propagated to the next time step, together with the updated cell state c_t. Right: GRU, similar to the LSTM unit but without cell state.

2.7 Regularization

A potential difficulty with machine learning algorithms is finding a good generalization, i.e., low loss not only on the training set but also on previously unseen data.

If a network does not generalize well to other data but has low error on the training set, it *overfits* to the training data. In order to prevent overfitting, a number of different so called *regularization* techniques can be applied.

A common mistake when designing network architectures is creating too big and complex networks with too much freedom. With enough parameters, a model might describe any dataset perfectly, yet still fail to generalize well to unseen data. In that case it rather ”remembers” the data than finds the most important features of the data. On the other hand, big networks tend to be more powerful, making network size a trade-off.

One common regularization method is *early stopping*, which simply means terminating the training before the network overfits [13]. In practice, the network should stop before the validation loss starts to increase, see fig 2.11.

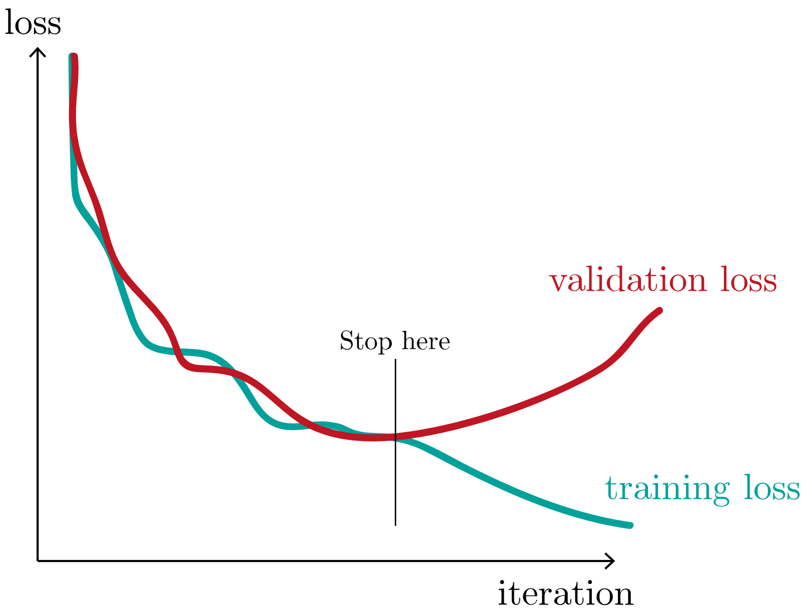


Figure 2.11: The validation error increases while the training error still decreases.

Note that it is essential the data is of sufficient quality. If the training data does not represent the real data fairly, it will be hard or even impossible to acquire good results. Naturally, if a network trained on data from one distribution is presented with data from a radically different distribution, the output is expected to be poor.

2.7.1 Dropout and Spatial Dropout

Dropout is a regularization technique which effectively accumulates multiple weaker models to produce a stronger model, i.e., with better predictions. However, training multiple networks could require an impractical amount of time and computational power. Instead, dropout seeks to approximate this process by *dropping* some neurons in each iteration of the training to produce smaller networks within the large network, hence the name ”dropout” [23]. An illustration can be seen in 2.12.

Dropping a neuron is equivalent to setting its output value to zero. Each neuron is dropped with some probability $p \in (0, 1]$ and thus retained with probability $(1-p)$. This is equivalent to sampling a random network of subset neurons from the full network, for every training iteration. The training then progresses as normal, where the data is passed forward through the network, the error is propagated backwards and the parameters are updated. By only training a sampled network in each iteration, complex co-adaptions between neurons are broken. This means that the output of one neuron must not rely too heavily on the output of any other neuron, thus becoming more robust to noise. When the network has finished training, all neurons are enabled. Since every training iteration was performed with on average $(1-p)$ times the total number of neurons, the parameters now need to be scaled by a factor $(1-p)$. In effect, the expected output of each neuron is the same as during training, and the full network is an approximation of all possible combinations of subset networks [23].

Spatial dropout can be seen as the CNN-equivalence of standard dropout. With knowledge of the original dropout approach explained above, it might feel sensible to drop neighbouring neurons. However, this idea will not work well in the context of images, as nearby pixels in an image are usually highly correlated. Adjacent pixel can even be said to have approximately the same value. By the same principle, error gradients from adjacent pixels contribute with approximately the same value. By dropping one of the two neighbouring pixels (neurons) as when applying standard dropout, the total contribution of gradients is effectively halved. As noted by [24], a factor p of all neurons are dropped on average, meaning that training is slowed down by a factor p without any gain in performance. Instead of using the original dropout approach, [24] suggests dropping entire feature maps and show that performance is increased by doing so. Dropping entire feature maps, or channels, is called *spatial dropout* and retains the original idea introduced by [23].

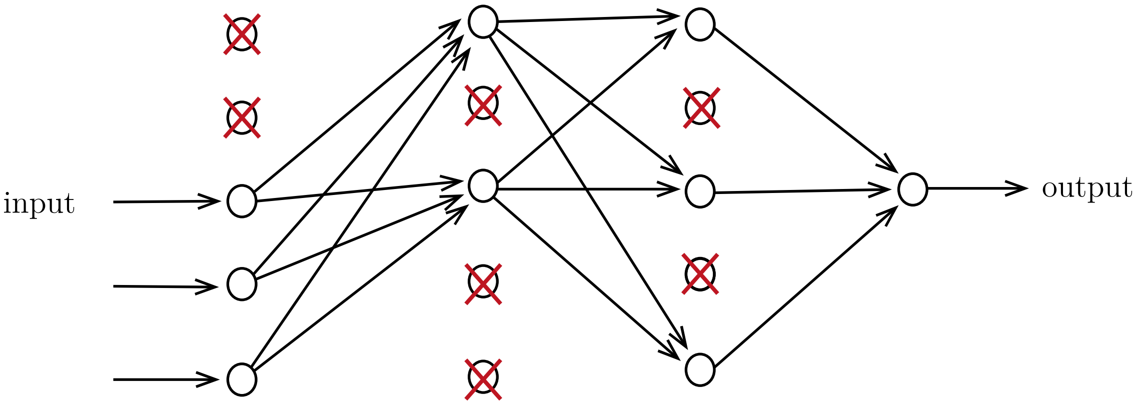


Figure 2.12: An illustration of dropout. The crossed out neurons are dropped and not used for this forward and backward pass.

2.7.2 Weight Decay

Another typical regularization method is *weight decay*, also known as L2 regularization. In other contexts L2 regularization is also called rigde regression or Tikhonov regularization. The main principle behind weight decay is to penalize the size of the free parameters (weights and biases) in the cost function $\tilde{L}(\cdot)$ [13]. Earlier, the cost function was the error function, as we simply wanted to minimize the error between predicted values and ground truth. But now we add a weight decay term, and thus want to minimize

$$\tilde{L}(f(\boldsymbol{x}, \boldsymbol{\theta}), \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^\top \boldsymbol{w} + L(f(\boldsymbol{x}, \boldsymbol{\theta})\boldsymbol{y}). \tag{2.29}$$

By adding the L2 term, the magnitude is maintained mostly for parameters contributing significantly to reducing the loss. Conversely, the magnitude of parameters which contribute less to decreasing the loss will be heavily reduced due to the penalization. As a result, some of the parameters will decay towards zero, thus limiting the chances of overfitting to the data [13]. In other words, weight decay prevents the parameter vectors from growing unnecessarily large and limits less significant weights from pointing in an arbitrary direction, otherwise dragging down the performance and generalization [12].

2.8 Control module

The path which is output from the network is used to calculate actual steering signals to the car. There are in general three ways to do so, namely using path stabilization, trajectory stabilization or model predictive control (MPC) [25]. While all methods are used to reach a desired position along a given path, path stabilization disregards the time for reaching each position. Because control over the acceleration is a requirement (in order to start and stop) and safety constraints are desirable, the MPC is preferable in this project and is explained below in Section 2.8.1. For the interested, the theory behind the path stabilization technique *pure pursuit* is covered in Appendix B.

2.8.1 Model Predictive Control

For each time step, a reference trajectory is planned for a number of future time steps, using the current state of the vehicle and information about its surroundings. A series of future steering signals are optimized as to result in the reference trajectory (according to some cost function) while still being within the physical limits of the vehicle. The first steering signals of the output optimized series are applied, and the process repeats. Another example of such a controller is the *model predictive path integral* (MPPI) controller used in [5].

The controller performs simulations on a kinematic bicycle model. The model is defined, as in [26], as a vehicle with one front wheel and one rear wheel, positioned at distances l_r and l_f form the center of mass, see Fig. 2.13.

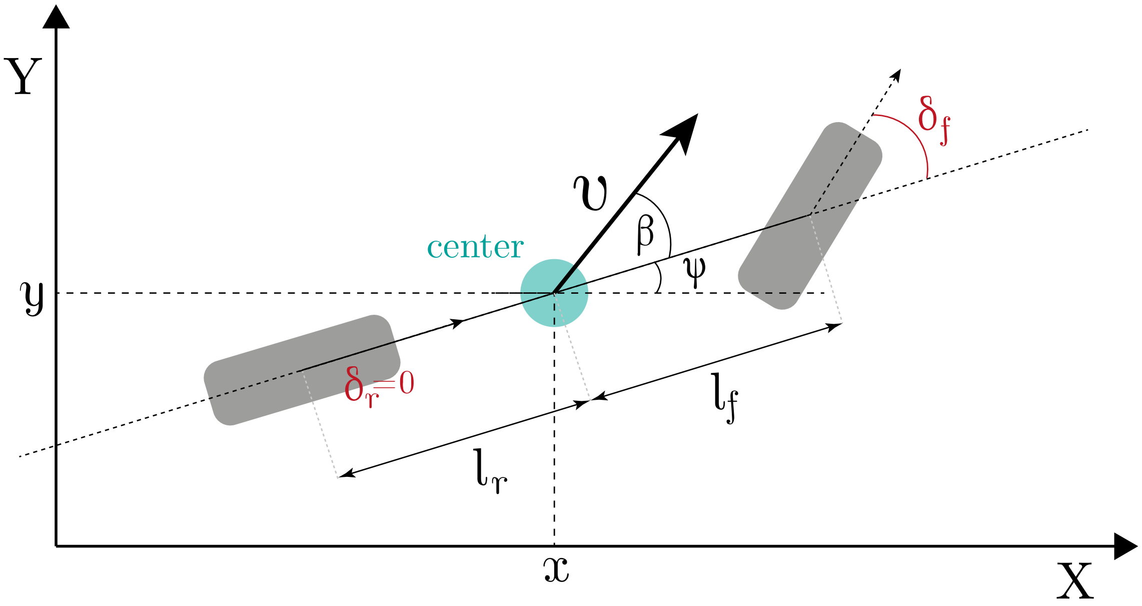


Figure 2.13: A kinematic bicycle model of a car.

The state S is defined as

$$\begin{aligned}
\dot{x} &= v\cos(\psi + \beta) \\
\dot{y} &= v\sin(\psi + \beta) \\
\dot{\psi} &= \frac{v}{l_r}\sin\beta \\
\dot{v} &= a \\
\beta &= \tan^{-1}\left(\frac{l_r}{l_f + l_r}\tan\left(\delta_f\right)\right),
\end{aligned} \tag{2.30}$$

where x and y are the coordinates of the center of mass, ψ is the heading of the vehicle in the global frame, v is the velocity and δ_f is the steering wheel angle at time step t. Like any ordinary car, the back-wheels are not used for steering and thus it is assumed to always have $\delta_r = 0$. The dotted variables represent the updated values at time step $t + 1$.

The control inputs to the MPC, i.e., the variables used to control the kinematic bicycle model when simulating driving, are the acceleration a and the steering wheel angle δ_f [26]. Applying a series of control inputs to some initial vehicle state yields a trajectory. The distance between the resulting trajectory (for example using MSE) and the desired trajectory can be minimized by tuning the control input series, using some optimization technique.

In order to properly model the dynamics of the CARLA vehicle, a neural network was used. The neural network aims to find the relationship between acceleration and the throttle and brake signals of the particular car model used in the simulator.The neural network input consists of the simulated car's state, defined as the steering angle, heading angle, forward speed (in the direction of the vehicle heading) and a desired acceleration in the direction of the vehicle heading. Given this state, the network predicts the throttle and brake signals to apply in order to achieve the desired acceleration. When the MPC has produced a control input series, the acceleration a can be mapped to actual throttle and brake values using this network.

Furthermore, as the MPC constructs the control signals by optimization, additional constraints can be included. In this project, a representation of the road boundaries was included in the form of a distance map for each obstacle. Each road map was first divided into a grid, for which the distance to the nearest road boundary was calculated, together with the corresponding gradients of the distance w.r.t. the grid cell position, see Fig. 2.14. The distance maps were calculated using road maps, provided on the CARLA github site[1].

Each distance map was calculated by first finding the perimeter pixels of the corresponding object in the road map image (top image in Figure 2.14). For each pixel in the whole image, the smallest distance to any of the perimeter pixels was set as the pixel value in that particular distance map. Any point in the map can be queried for the (bilinearly inerpolated) distance and the gradient of the distance, which are both used by the MPC.

2.9 Clustering Algorithm

One of our ideas are based on a classification network and in order to first extract the different classes, we need to use a clustering algorithm. The application and context of the clustering algorithm is explained in Section 3.3.3, however in this section we will explain how the algorithm works. There are several approaches to cluster data, we use *k-means clustering* which is applicable for data of arbitrarly high dimension.

2.9.1 K-means

K-means is an iterative clustering algorithm which aims to divide the N data points into K clusters and find their respective mean value $\boldsymbol{\mu}_k$. A data point $\boldsymbol{x}_n$ belongs to the cluster k which minimizes $(\boldsymbol{x}_n - \boldsymbol{\mu}_k)^\top(\boldsymbol{x}_n - \boldsymbol{\mu}_k)$, i.e., the cluster with closest mean. After all $\boldsymbol{x}_n$ have been assigned a cluster, the means $\boldsymbol{\mu}_k$ are moved to better represent the clusters, using the formula

$$\boldsymbol{\mu}_k = \frac{\sum_n z_{nk}\boldsymbol{x}_n}{\sum_n z_{nk}}. \tag{2.31}$$

The full algorithm, as described in [27], is as follows

1. Initialize the cluster means to random values. Initialize the binary array z of size $N \times K$.
2. For each data point $\boldsymbol{x}_n$ find cluster k which is the closest. Set z_{nk} to 1 to indicate which cluster x_n belongs to. Set $\forall z_{nj} = 0$ for $j \neq k$.

[1] `https://github.com/carla-simulator/carla/issues/129`

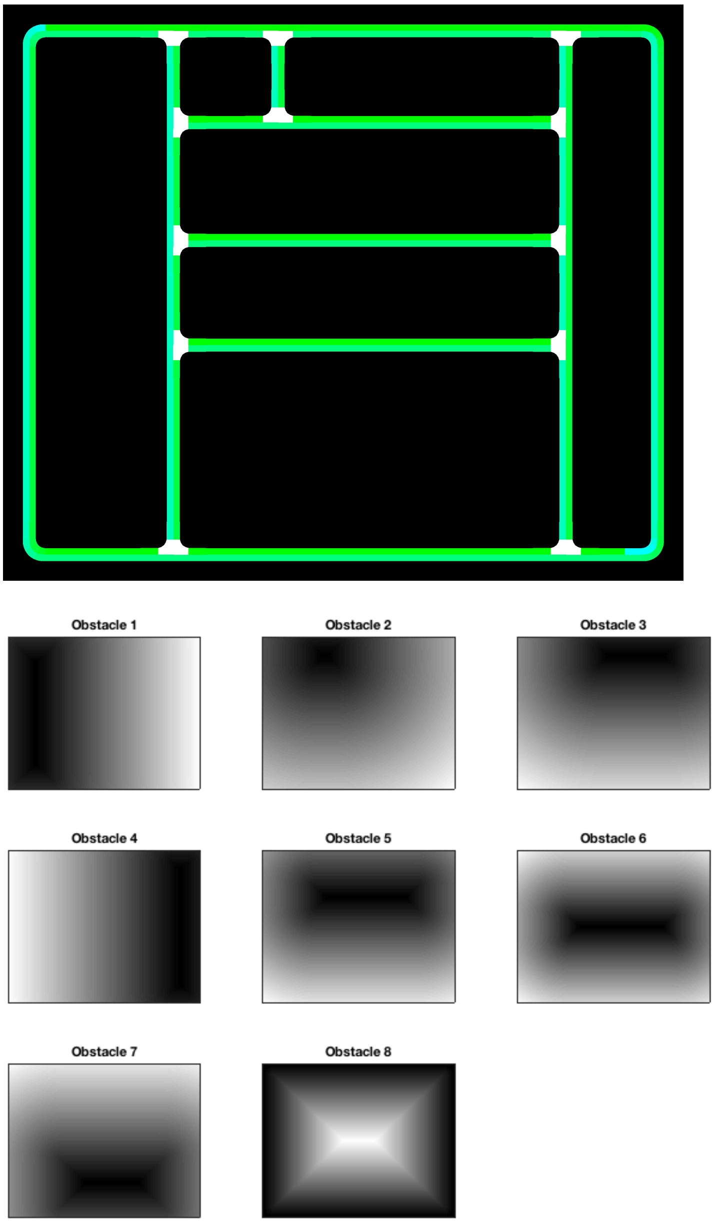


Figure 2.14: Top: road map with green and white drivable areas and black non-drivable areas. Bottom: Distance map of road boundaries to all non-drivable areas of the town. Black means small distance and white means high. For example, Obstacle 2 represents the smallest area at the top. Each distance map corresponds to a constraint, telling the MPC not to allow actions that drive the car into the non-drivable areas.

3. If z is unchanged from the last iteration, i.e., the clusters are unchanged, stop.
4. Update each $\boldsymbol{\mu}_k$ according to eq. (2.31).
5. Go to 2

In summary, the algorithm aims to minimize the total distance D, calculated as

$$D = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk}(\boldsymbol{x}_n - \boldsymbol{\mu}_k)^\top (\boldsymbol{x}_n - \boldsymbol{\mu}_k). \tag{2.32}$$

This algorithm will always converge to a minimum. However, finding a global minimum is not guaranteed as the initial weights are random. In order to find the global minimum one would need to find all combinations of clusters for all data points which quickly becomes unfeasible. In practice, one runs the algorithm several times and then picks the clustering with lowest D [27]. An example of a finished k-means clustering is seen in Figure 2.15.

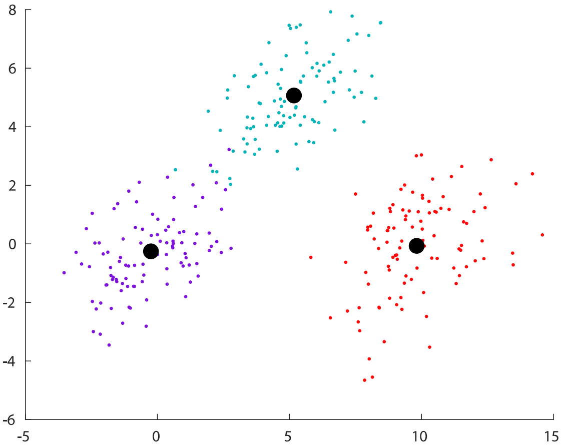


Figure 2.15: Example of data clustering. The black points is the μ_k (the centroids) of respective cluster.

2.10 Principal Component Analysis

Principal Component Analysis (PCA) is commonly used for dimensionality reduction of data, and is described below in accordance with [27]. PCA can be beneficial for, e.g., feature selection and visualization of data. PCA projects data from m to d dimensions using a linear combination

$$x_{nd} = \boldsymbol{w}_d^\top \boldsymbol{y}_n \tag{2.33}$$

of N data points $\boldsymbol{y}_n$ (of dimensionality m) and m vectors $\boldsymbol{w}_d$, where d is the number of desired dimensions. In other words, each point is converted from $x_{n1}, .., x_{nm}$ to $x_{n1}, .., x_{nd}$. The algorithm aims to find the best projection vectors $\boldsymbol{w}_d$, i.e., which maintain the most data in the lower dimensions. The vectors $\boldsymbol{w}_d$ are also called the *principal components*.

The principal components $\boldsymbol{w}_d$ is chosen based upon which projection results in the highest variance of the data. By looking for the highest variance, the most distinguished features of the data is maintained. After finding the first principal component $\boldsymbol{w}_1$ the following $\boldsymbol{w}_2$ must be orthogonal to $\boldsymbol{w}_1$, i.e., $\boldsymbol{w}_1^\top \boldsymbol{w}_2 = 0$. The third component must then be orthogonal to both $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$, and so on. An intuitive example of the usage of PCA is illustrated in 2.16.

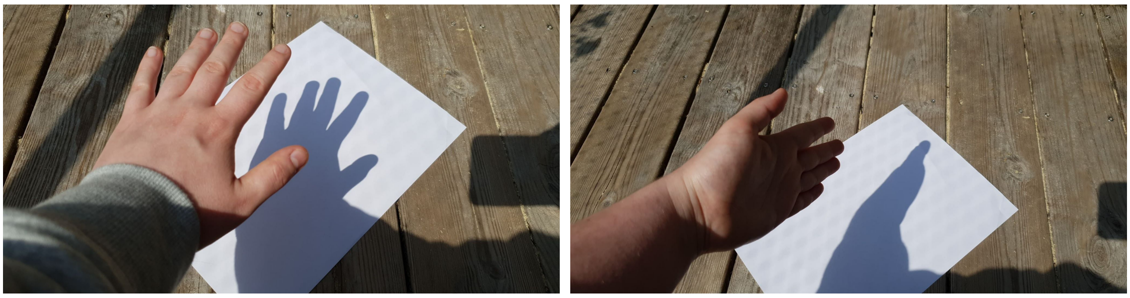


Figure 2.16: The hand is reduced to 2 dimensions in the shadow. By picking projection vectors $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ with the most variance the features are retained in the dimension reduction (left image). Choosing principal components poorly can cause the projected data to be unrecognizable (right image)

In this project PCA is used in one of our main ideas. By lowering the number of variables the network is expected to output, we theorize the network has easier to learn. More about the usage of PCA in Section 3.3.2 and 3.3.3.

2.11 Semantic Segmentation

Semantic segmentation is an image processing problem in which one wishes to find and identify objects on a pixel level. Each pixel is assigned to one of the predefined semantic classes. A good semantic segmentation system will classify each pixel correctly, and thus create a *semantic segmentation map* where image regions are partitioned with semantic coherence, yet with precise boundaries. A binary class example of a semantically segmented image can be seen in Figure 2.17.

There are many ways to tackle this problem, but the most successful approaches are based on deep *fully convolutional neural networks* (FCN) which perform a pixelwise class prediction [28, 29, 30]. An advantage of using a FCN is that the input image size is unrestricted. Furthermore, in contrast to general deep neural networks which compute a general nonlinear function, a FCN calculates a nonlinear *filter* [28].

In the area of autonomous driving, semantic segmentation is frequently used for environment perception and applied on the output of, e.g., cameras. The image-feed is processed through a semantic segmentation filter and the output is typically forwarded to an action decision system. For instance, [31] use semantic segmentation in this fashion.

The usage of semantic segmentation in this project will differ from the ”classical” idea where some objects in an image are located and identified. Instead, this project will use semantic segmentation in a similar fashion as [6], that is, using a pixelwise classification to predict which pixels the car will be located on in the future. In contrast to [6], the model presented in this report (Section 3.3.4 will also consider at which relative time step each position is reached, consequently forming a trajectory.

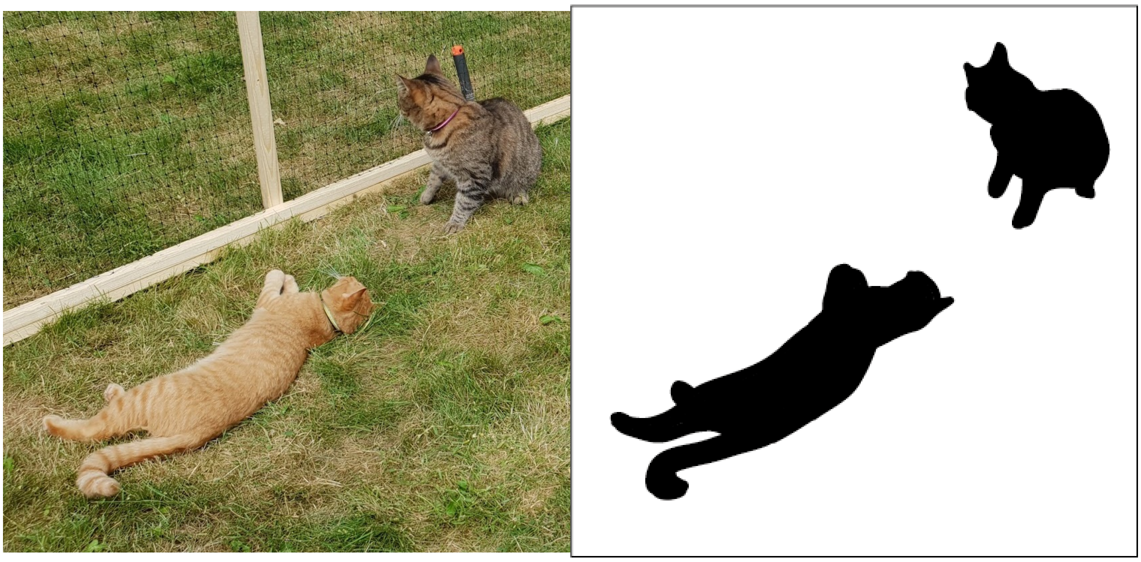


Figure 2.17: To the left is the original image and to the right is the semantically segmented equivalent. The problem is binary, with only two classes: ”cat” and ”not cat”.

2.12 Object Classification, Localization and Detection

The problem of producing a desired trajectory of a vehicle, as formulated in this project, is somewhat related to other computer vision tasks. Such tasks may include *object classification*, *object localization* and *object detection*. The following paragraphs will briefly explain said problems, some proposed solutions to them, and how they relate to the path prediction problem.

Object classification is the problem of determining which category a single object in an image belongs to, i.e., what the image as a whole depicts. A probability is assigned to each of the possible categories, where the one with highest probability may be used as the final prediction. In contrast, object localization refers to the problem of finding the spatial location of an object in an image. This is often done using a square bounding box which encapsulates the whole object. In conjunction with object classification, it is possible to both locate and classify not only one object, but a fixed number of objects within an image. A more general problem is that of object detection, which seeks to locate and classify all object instances (from the possible categories) appearing in an image. This means that the number of classified objects may be different from image to image [32], as well as what classes are found.

One of the suggested solutions to the object detection problem incorporates so called region proposals, which indicate regions in the image that are likely to contain an object of interest. The proposals may be inferred using some standalone algorithm as in [33] or using an end-to-end model which predicts both region proposals and class probabilities, as in [34]. In the case of [34], a fixed number of regions, or bounding boxes, of varying size are distributed evenly over the input image. For each bounding box, the probability of containing an object is predicted, as well as proposed adjustments to the bounding box as to yield a better fit around the object. Finally, the adjusted bounding boxes are used to extract regions of interest from the input image. The regions with high enough probability of containing an object can then be classified.

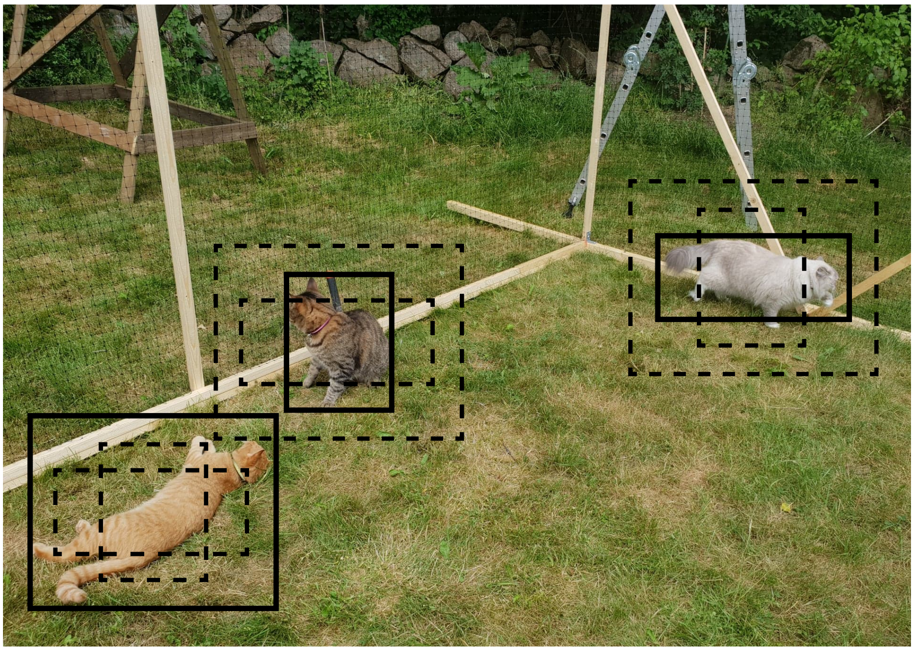


Figure 2.18: Nine bounding boxes of three sizes at different locations. Notice how the best fit around each cat is achieved using different box sizes. Adjusting the height, width and location would potentially make even better fits.

One of the most interesting parts of the described system is that it performs both classification (i.e., region does or does not contain an object) and regression (i.e., adjustments to location, width and height of bounding box) simultaneously. With respect to this project, classification could be done over different types of trajectories, e.g., left turn, right turn, constant forward velocity or accelerating forward velocity. Trajectory classes could be found using a clustering algorithm such as k-means. Regression could then be performed to construct adjustments to a typical trajectory of the predicted class, e.g., elongating a left turn or shorten a forward trajectory with constant velocity. A parameterization of trajectories could be done using PCA.

When performing object classification and localization in conjunction, one could also imagine a system which simply predicts a class and one bounding box (or adjustments to some predefined bounding box). In that case, the system would be class agnostic, meaning that the bounding box is produced using the same model parameters regardless of which class is predicted. In contrast, a class specific system would produce the bounding box using class specific parameters [32]. This can in practice be implemented by predicting individual parameters for each class, but computing the loss only for the ones corresponding to the predicted class. For example, a class specific system with c classes and four bounding box parameters (width, height, x- and y-location) would produce $c \times 4$ parameters, while a class agnostic system would produce only $c \times 1$.

When locating an object in an image, the predicted object class might not influence the properties of the bounding box more than, e.g., the distance and perspective do. However, when adjusting a trajectory, the proper adjustments are highly dependent on which type of trajectory is to be adjusted. For example, a right turn trajectory which perfectly aligns with the ground truth is not be adjusted at all, while a left turns should be mirrored. Hence, it is crucial to construct a class specific model in this project.

A final note on the theory of class specific object classification and localization models: the error of the predicted adjustments should only be backpropagated for the ground truth class, as in [35]. As the adjustments are class specific, it would be meaningless and most likely hurt performance severely to compute the loss on the incorrect classes as well.

3

System Overview

This chapter will give a brief overview of the system and how it is structured. Sec. 3.1 will describe the overall program and the most important components. The system is built to drive a car (or a simulated car) continuously, i.e., online. However, the supervised network is trained in an offline setting with prerecorded data and ground truth. A general overview of how to train a network is covered in sec. 3.2. Lastly the three main ideas for the neural networks will be presented in Section 3.3

3.1 Online system (test driving)

CARLA is used to simulate a car and continuously receives steering signals. A rough UML sequence diagram of the system can be seen in fig. 3.1. Note that the diagram is not exact, rather, clarity is prioritized.

The simulation of the virtual world and the cars is called server. The server is provided by CARLA and we have not changed this except for settings and a few small modifications to better suit our model. The client is the program which runs the machine learning algorithm and the decision making. The client is provided with all the data from the server, e.g., lidar, GPS-IMU and traffic status. The client is then expected to return steering signals for the car to the server. Unlike in reality, the time in the simulator is bound to discrete steps. One time step in the simulator is further referenced to as a *frame* or *time step*, and the update rate is 10 frames per second.

After the client receives the data and saves it, a preprocessing step is initiated. The preprocessing converts and stacks the data to match the desired form of the input to the network. The network calculates the desired path which is then fed to the MPC module. The MPC is also provided with the current position of the car, its velocity and its heading angle. From the MPC, the car's next steering angle, brake and throttle is returned and forwarded from the client to the server, which executes the actions. The data for the next time frame is then provided to the client and the loop continues until the simulation is halted.

3.2 Training a network

The main focus of this project have been to develop a good network which outputs a good trajectory prediction for the car. The system explained above uses a fully

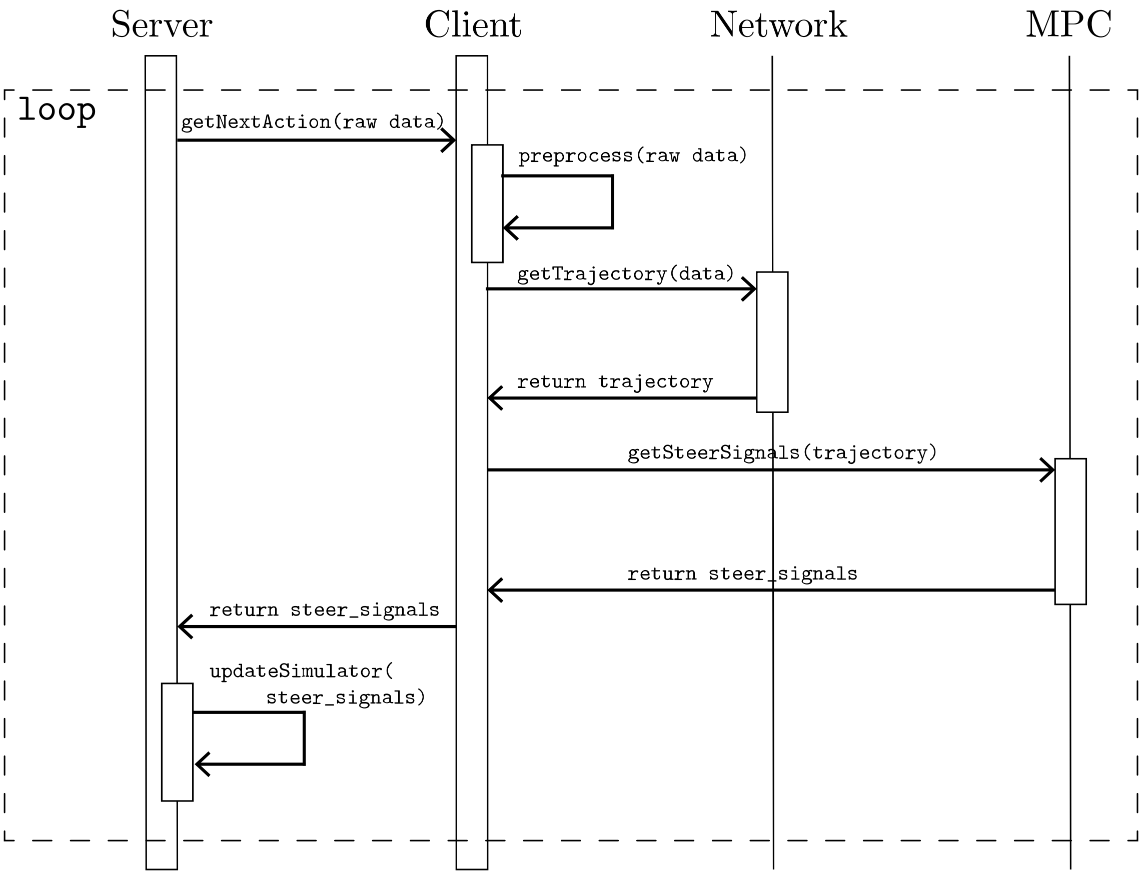


Figure 3.1: A simplified sequence diagram of the system.

trained network used only for inference, and which is trained in a separate offline program.

Firstly data is gathered and saved to disk using the CARLA server and the provided autopilot client. The autopilot is mildly modified to drive zigzag in order to get more varied data and to make the model robust to small driving errors. When the data is collected, it is annotated and driving intentions are generated. The rest of the preprocessing occurs and the updated data is saved. Now the network trains on the data, and the ground truth for the current time step t is fetched from the n_f future time steps $(t+1, t+2, ..., t+n_f+1)$.

The model is saved regularly during training, both as a safeguard if the training crashes and as a means to prevent overfitting. The validation- and training loss is also saved regularly to facilitate an analysis of its performance. When finished, the network is evaluated on the test set and all results are saved. If the results look promising, the best performing iteration of the network can be used in a driving test in the system described above in sec. 3.1. Note that the MPC module is not used in the training and (obviously) no feedback is given to the server as the network trains offline.

3.3 Our Main Ideas

During the project, many neural networks have been trained, tested and evaluated. Some neural networks have only minor differences, such as variations of hyperparameter values, while others differ on a more fundamental level. There are three main neural network ideas that have been examined in this project, each tested with different hyperparameters values. The ideas will be briefly introduced in the following sections to provide a better overview of the project.

3.3.1 Pure Regression Idea

One of the most basic ideas for predicting a trajectory is to simply define a regression problem from some input to an output sequence of x- and y-coordinates. We refer to this idea as pure regression, since no classification or parameterization of the trajectory is made. One advantage of using this definition is that it is straight forward to implement. Moreover, highly complex trajectories can be formed, as there is no restriction on the relationship between coordinates in the sequence. In other words, the shape of the trajectory could have arbitrarily sharp turns in any direction. However, giving the regression model this freedom could also be a disadvantage, as predicted trajectories may become rough and hard to follow in practice.

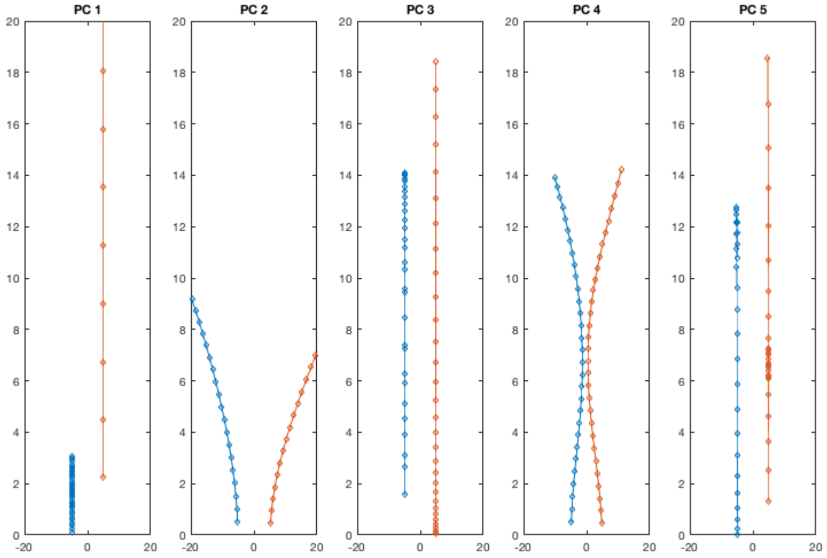


Figure 3.2: Trajectory reconstruction using one of the five leading principal components of the training data, respectively. Blue lines correspond to low (possibly negative) principal component values, while orange lines correspond to high values. From left to right, the characteristics could be described as *speed*, *right turn*, *acceleration*, *minor right turn* and *slow down, then speed up*. Each line has been offset slightly on the horizontal axis for clarity, while vertical offset is just part of the principal component. Axis unit is meters.

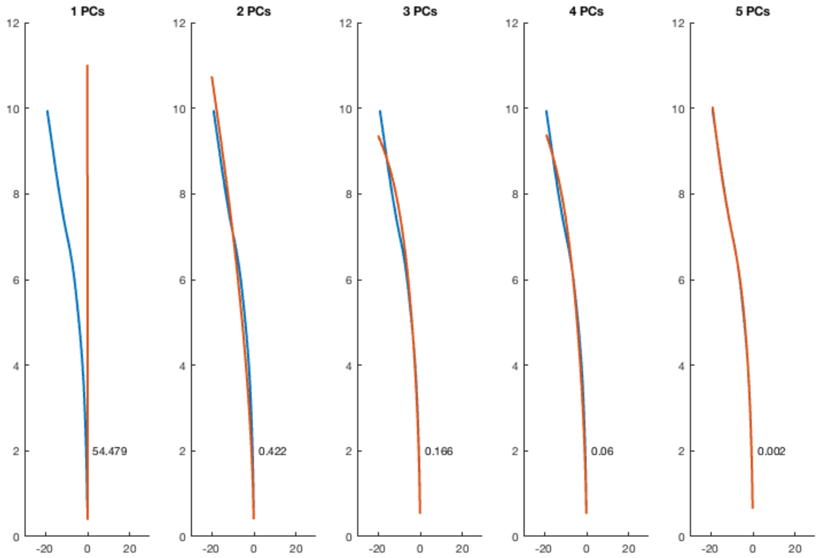


Figure 3.3: Trajectory reconstruction (orange) made by accumulating one through five of the leading principal components of the training data. The reconstructed trajectory aligns better and better with the original trajectory (blue) for each added principal component. The MSE between curves is displayed in the lower part of each plot. Axis unit is meters.

3.3.2 PCA Regression Idea

In contrast to predicting each coordinate explicitly, a trajectory can be formed by predicting its overall characteristics. Speaking in broad terms, such characteristics could be the amount of acceleration, amount of turning, turn direction, etc. These characteristics could then be used as parameters of a function which in turn outputs a sequence of coordinates. Compared to directly predicting each coordinate, predicting the overall trajectory traits could lower the number of output variables. More interestingly, the shape of the trajectory would by default follow a smooth line which is then modified according to the predicted characteristics. This restricts the trajectories and ensures smoother driving paths, more similar to our ground truth.

Instead of creating a function by hand which produces trajectory coordinates from some defined characteristics, important traits of the recorded trajectories can be found using PCA, see Figure 3.2. The predicted principal components can then be used to create a sequence of coordinates, as described. An example of this can be seen in Figure 3.3, where it is shown that the five leading principal components can be used to reconstruct the example trajectory with an MSE of 0.002.

3.3.3 Cluster Idea

Predicting the trajectory can also be posed as a classification problem, where each class corresponds to one type of trajectory. As opposed to the separate characteristics described in 3.3.2, one trajectory class would represent a specific combination of characteristics, such as slow left turn or large acceleration straight ahead. Different classes of trajectories can be found using the k-means clustering algorithm, as explained in 2.9.1. The centroid of each cluster corresponds to the typical trajectory of that class, see Figure 3.4.

The class-typical trajectories are fixed and it may be that none of them describes the ground truth trajectory very well. As a solution to this problem, adjustments to the trajectories' principal components are predicted as well. This enables the model to compensate for differences between class-typical and ground truth trajectories.

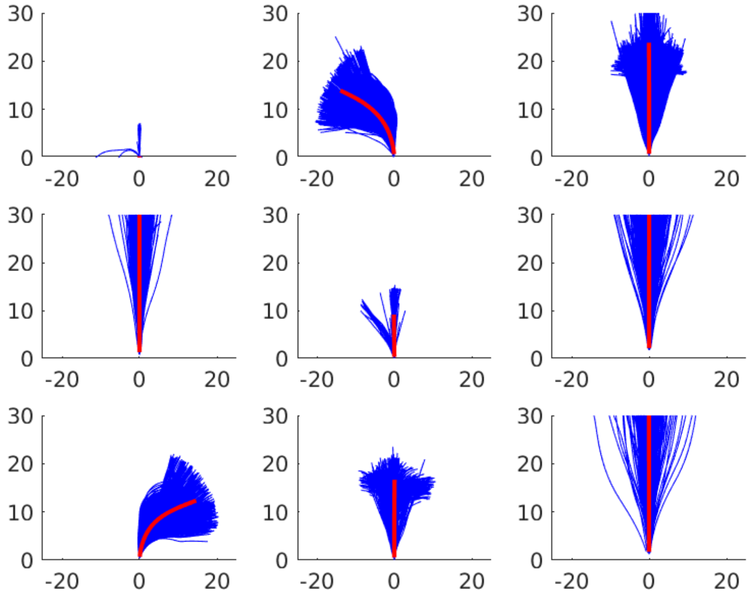


Figure 3.4: Nine clusters formed using the k-means algorithm on the training data. Blue lines are observed trajectories, while the red line in each cluster is the corresponding centroid. Axis unit is meters.

3.3.4 Semantic Segmentation Idea

The problem can also be formulated in terms of a semantic segmentation problem, and thus remove all regression from the problem. More precisely, it is a binary pixel-level semantic segmentation problem and thus only two classes are used in this idea: ”Part of the trajectory” and ”Not part of the trajectory”. The binary classes will be encoded 1 and 0 respectively. In the output of the network, which is an image, each pixel has a probability of it being part of the future trajectory or not. An image with probabilities will be produced for each future time step one wishes to predict. For example, for 6 future time steps the output will consists of 6 images each of same size, in which the pixels will have a probability of being a part of the trajectory for that specific time step.

To fit this type of classification output, the ground truth must be customized. For each future ground truth coordinate, an image of probabilities is created. Around the coordinate a circle is drawn, and all pixels within the circle is set to 1 and all pixels outside the circle is set to 0. The circle have a quite big radius in order to provide the network with enough positive feedback for it to learn within a feasible time. An example of the generated ground truth is shown in Figure 3.5, where each image corresponds to an output channel, i.e., time frame. The motivation behind this idea is its similarities with the method in [6], which seems to produce good results. The major differences between [6] and this approach is that time needs to be taken into account here.

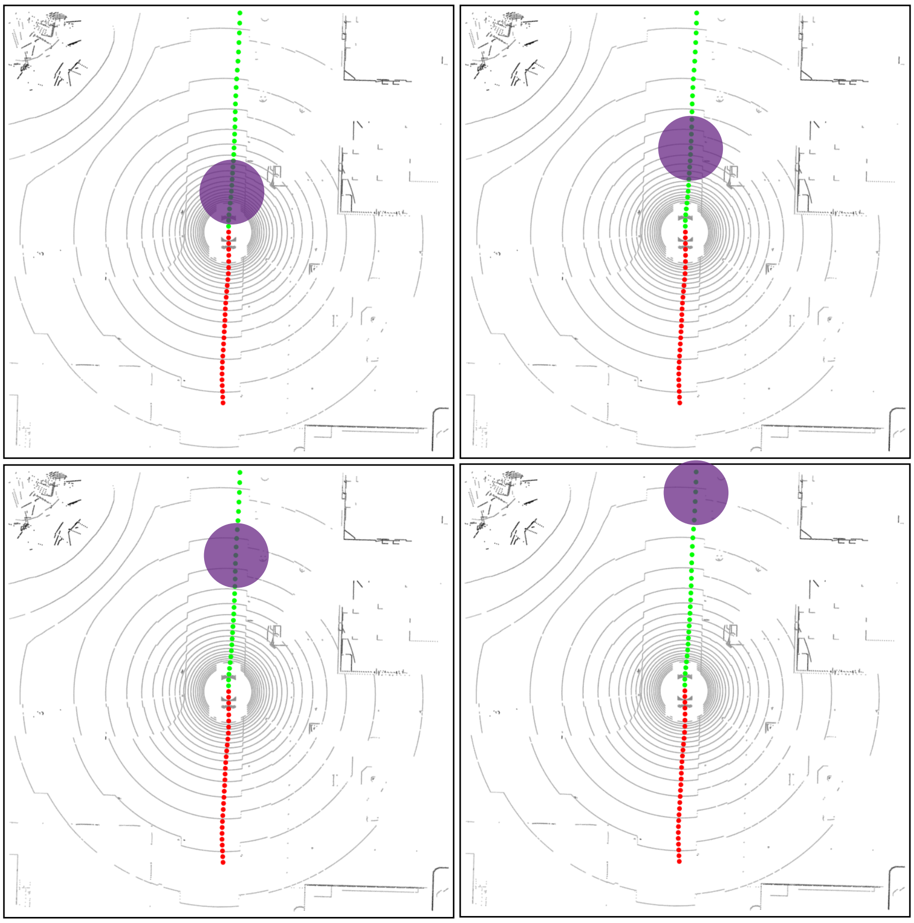


Figure 3.5: The red dots is the past path, the green small dots are the recorded ground truth trajectory. The purple circles are the generated ground truth for the semantic segmentation. Each image corresponds to one channel, in other words one specific future time frame.

4

Method

This chapter will recount the method used in the project. In general, the project has been implemented in an iterative fashion, starting with a smaller system which has been expanded upon. However, no established agile method was used.
The first section will describe the tools and software used. Section 4.2 will depict how the data was collected, and is then succeeded by a description of the data preprocessing process in Section 4.3. A detailed account of the neural network architectures are given in Section 4.4. The implementation of the MPC is briefly explained in Section 4.6 and the chapter is concluded with an description of the neural network evaluation method in Section 4.7.

4.1 Tools and Equipment

The hardware for this project consisted of one PC with a quad core Intel Core i5-7600 @ 3.5 GHz, with 32 GB DDR4 RAM. The graphics card was an Asus NVIDIA GeForce GTX 1080Ti with 11GB memory capacity.

The used software included linux system with CUDA[1] 9.1 and Pytorch[2] 0.4.0. Python[3] 3.6 was mainly used, but a few libraries required an older version. Lastly, after some research, CARLA was the preferred simulator for the project.

4.1.1 CARLA

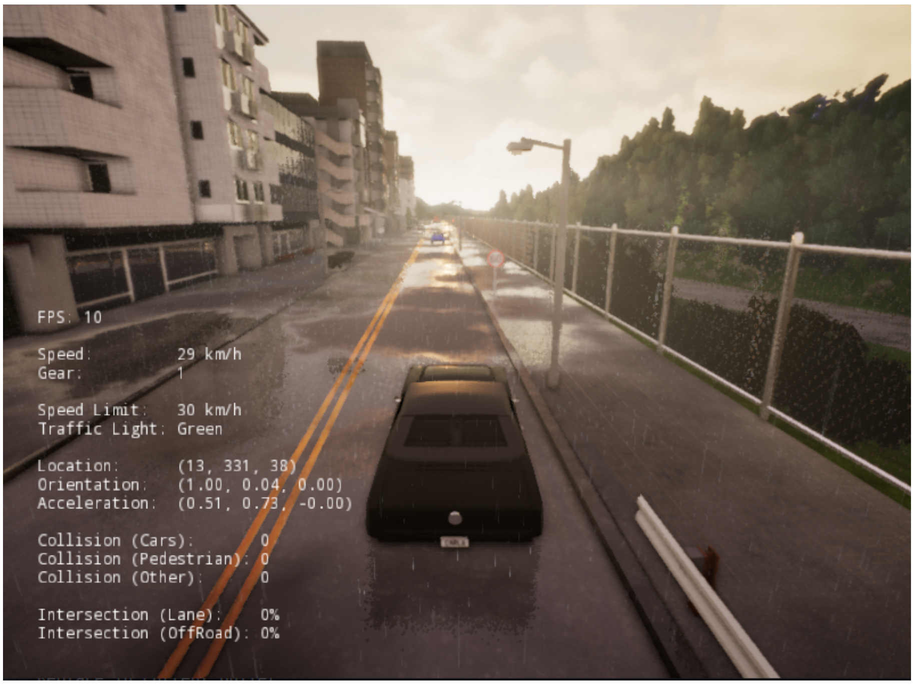
Numerous simulators were compared to find the best choice for this project. The free open-source options are few and the available simulators were AirSim (by Microsoft) and CARLA. None of the simulators supported LIDAR at the time of the decision. Due to the tight time schedule, CARLA was preferred as it provided preexisting towns and benchmarks, and it also seemed easier to use. An image from the CARLA simulator is shown in Figure 4.1.

It is easy to create one's own autonomous car and test drive it in two different preexisting virtual towns, Town01 and Town02. CARLA can provide a feed from cameras which features RGB-values, depth-values and also online semantic segmentation. The car simulations are based on NVIDIAS PhysX implementation of ve-

[1] https://developer.nvidia.com/cuda-downloads

[2] https://pytorch.org/

[3] https://www.python.org/



```
FPS: 10

Speed:          29 km/h
Gear:           1

Speed Limit:    30 km/h
Traffic Light:  Green

Location:       (13, 331, 38)
Orientation:    (1.00, 0.04, 0.00)
Acceleration:   (0.51, 0.73, -0.00)

Collision (Cars):        0
Collision (Pedestrian):  0
Collision (Other):       0

Intersection (Lane):     0%
Intersection (OffRoad):  0%
```

Figure 4.1: An image from the CARLA simulator.

hicles. CARLA also had numerous examples on how to use the software and clear and easy to follow tutorials. As of spring 2018 it also provides the possibility to use LIDAR sensors. The biggest drawback of CARLA was the late implementation of LIDAR and the software is computationally heavy, therefore the simulations is time consuming. Before the release of the lidar implementation we gathered the lidar values using CARLAs depth maps, which is documented in appendix A.

4.2 Gathering data for training

Data was gathered by recording sensor values in the simulator with a frame rate of 10Hz. We recorded all state-values of the car, e.g., acceleration, yaw, velocity, throttle, brake, etc. We also recorded the state of relevant surrounding objects, e.g., position of traffic signs and lights as well as the state of the ligths. A full list of recorded data can be seen in appendix C. The simulated lidar was set to mimic a Velodyne HDL-32E. It consists of 32 beams which is approximately evenly distributed in the vertical range from -30.67 to 10.67 degrees.

CARLA's autopilot was used to drive the car, as it could be left to perform the time consuming job while following the traffic rules. The autopilot decides the route randomly and cannot be configured beforehand. Uniform noise was added to the steering signal of the autopilot, making the car zigzag a bit. This was done in hope that the networks would learn to correct for bad heading angles.

The data was collected in a number of *episodes* (i.e., periods of driving) in the simulator. Train, test and validation sets were created by recording 20, 8 and 1 episodes respectively. Each episode in the train set lasted for 18000 time steps (30 min real time). However, the car would often get stuck at some point for the remainder of the episode, due to unexpected behaviour of the simulator (other cars tipping over in a turn or pedestrians standing in the road). The unusable data was trimmed from the data set and the rest of the episodes (for the test and validation sets) were recorded for only 9000 time steps each (15 min real time) as a preventive measure.

The train set was recorded in the largest of two simulator towns (Town01), while the validation and test sets were recorded in the smaller one (Town02). This allowed for training on more diverse scenarios, while reducing the correlation between train and test data.

4.2.1 Gathering Intentions

In order to create intentions for the data sets, the driving path needed to be known. However the autopilot drives randomly through the town, so manual annotations of the path were created in retrospect. With a script the annotations are converted to intention direction and intention proximity values and saved with the rest of the data.

Furthermore the traffic information is also documented and saved. CARLA saved the position of all traffic lights and traffic signs and also records the status of the traffic lights. A script saves the relevant traffic information by checking which traffic object the car passes. The saved traffic information includes the current and upcoming speed limit, the state of the next traffic light, and the distance to the next traffic object.

4.3 Data Preprocessing

Before sending the data to the network, some preprocessing is necessary. This was as to reshape and structure the data but also to reduce the size of the data, as for example lidar point clouds are very information dense. Furthermore the data was very disproportionate between, e.g., right turns and left turns. This could cause the networks to form a bias to a certain action, therefore the data set was balanced during training. By enabling resampling, data from categories with fewer examples appear just as often as data from categories with a lot of data.
By enabling resampling, examples from small categories will appear more often, e.g. an example from category ”right” might appear 10 times on average, while data from ”straight” might appear only once.

4.3.1 Balancing the data set

A large amount of the train data consisted of passages on straight roads and waiting for traffic lights. In order to enable the networks to train more on critical scenarios, such as taking a turn without intention, the train data was manually divided into seven categories and balanced, see table 4.1. During training, each category was sampled with equal probability, except for category 'other'. This category mainly included undesired events such as crashes, and was not sampled at all. However, because the recurrent networks were trained in a sequential way, i.e., two consecutive mini-batches contained measurements from consecutive time steps, the balancing was not done for these networks.

Category	Frames	%
left	3903	1.7
right	2323	1.0
left intention	7735	3.3
right intention	5567	2.4
straight	103732	44.7
traffic light	108254	46.6
other	700	0.3

Table 4.1: Frame distribution over categories. Classes *straight* and *traffic light* constitute a disproportionate amount of all recorded frames.

4.3.2 Lidar

Each point cloud consisted of a number of real valued points in 3D space. This allows for taking a bird's perspective, as successfully used by [6], and is also used in this project. One advantage with this perspective is that distances to objects can be easier to determine in a top view compared to a horizontal view. The point cloud is trimmed to a 60x60 m square, centered around the lidar position. Remaining points are then divided into a 600x600 grid as seen from above the vehicle, corresponding to pixels in an image. The pixel value is based on the maximum elevation within the grid. The global maximum elevation is $7.65 \approx 8$ meter which can be calculated with the lidar scattering angle and the the field of visibility (60x60m). The sensor is placed in the center at 2 meter altitude. The pixel values of the max elevation where normalized to lie in $[0, 1]$ with the lidar positioned at 0.5. An image of the point cloud and the generated top view is visible in Figure 4.2. Other statistics could in theory be calculated and separated into different channels of the image. However, the maximum elevation should capture the most relevant information, such as road boundaries, other vehicles, pedestrians and so on. Note that other information normally extracted from a lidar, like reflectively, is not available in the CARLA simulator.

The maximum elevation image (lidar) l_t and vehicle measurements m_t made at time step t are collectively referred to as the state $\xi_t = (l_t, m_t)$ of the vehicle.

4.3.3 Rest of the Values

The extensive recorded "raw" data (example listed in Appendix C) is used to create a more comprehensive set of values. For the current time step t_c, the past n_p states

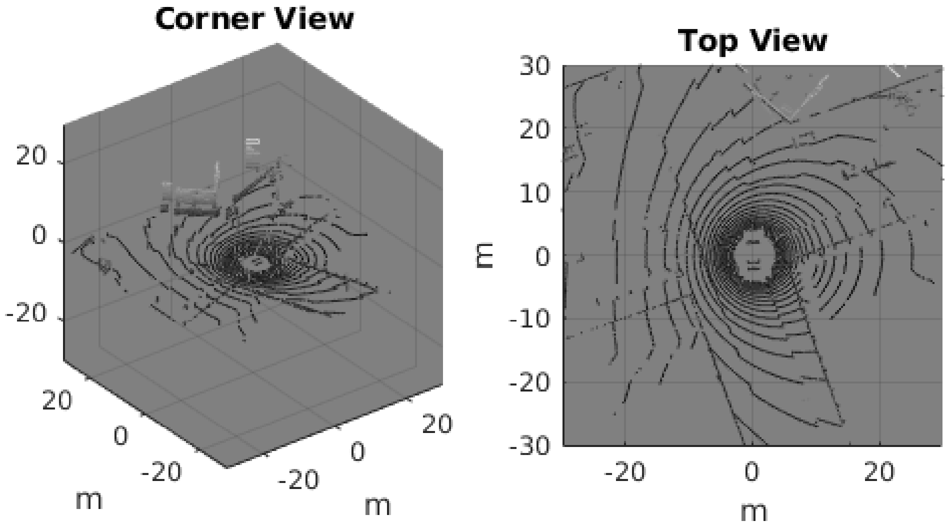


Figure 4.2: A 3D lidar pointcloud (left) is viewed from above (right). The lighter color represents higher altitude and darker color represents lower altitude.

ξ_t is used as input to the network. So for each time step t the measurement m_t is extracted, which includes the following values where

x_t, y_t is the coordinates relative to the cars position at t.

a_t is the total acceleration of the car at t.

s_t is the total speed of the car.

δ_t is the steering wheel angle.

intention direction is either left (-1) or right (1)

intention proximity is the distance in meters to an upcoming crossroads

speed limit at t is the maximum speed allowed

traffic object distance at t is the distance in meters to the next traffic light or traffic sign (whichever comes first).

traffic light status at t is the colour of the next traffic light. This is always known, no matter the distance to the traffic light.

If any of the last five values is unknown for any reason, it is set to 0. Additionally, there are two more values saved to m_t.

ψ_t is the yaw of the car w.r.t the global coordinates.

category is which category the time step belongs to, e.g., straight road, or right turn.

The last two values are not part of the input to the network, but are instead used for calculating the relative coordinates x_t, y_t and for balancing the data set, as described in Section 4.3.1. Note that the ground truth for t_c consists of the relative coordinates $\dot{p}_t = (x_t, y_t)$ for n_f future time steps, namely $\left(\dot{p}_{t+1}, ..., \dot{p}_{t+n_f+1}\right)$

In conclusion, the input to the network consists of the current state of the car $\xi_t = (l_t, m_t)$ concatenated with the n_p past states $\xi_{t-1}, ...\xi_{t-n_p-1}$. The values l_t and m_t might be introduced at different points in the network architecture, however both are always used. The ground truth trajectory, $\dot{p}$, is then compared with the predicted trajectory $\hat{p}$ through the loss function in order to train the network. Both $\dot{p}$ and $\hat{p}$ consists of n_f coordinate pairs (x, y).

4.4 Network Architectures

The basis of all networks has been the network proposed in [6]. It makes use of an encoder part, a context module and a decoder part. The encoder reduces the input size using convolutions and max poolings, i.e., *encoding* the input in a smaller sized tensor. The context module then makes use of dilated convolutions to obtain a receptive field covering the full encoded input feature maps, thus grasping the *context*. The decoder part then uses transposed convolutions, i.e., learned up-sampling, to produce an output image of the same dimensions as the input image, i.e., to decode the last feature map of the context module. Each pixel in the decoder output image contains an estimated probability of the vehicle driving over the corresponding position in the future. Since this project formulates the trajectory prediction problem in other ways, the decoder part is replaced by other parts, described in the following sections. Another noteworthy difference is that the non-lidar data from previous time steps in [6] are encoded in the form of image channels, whereas the data in this project is simply used as is. The encoder and context modules used in this project is listed in Table 4.2.

The network architectures listed below describe the overall differences between them. Variants of the same network architectures were trained, for example using different number of clusters or convolution channels. For more specific details on these architecture variants, see our github repository[4].

4.4.1 Pure Regression Networks

The following paragraphs describe the networks implemented for the pure regression idea explained in Section 3.3.

4.4.1.1 CNN Fully Connected (FCNet1, FCNet2)

A natural modification to the network proposed in [6] was to replace the decoder part with fully connected layers, to produce an array of future vehicle positions. As

[4] `https://github.com/AnnaNylander/exjobb/tree/master/network/architectures`

Layer	Type	Size	Channels	Stride	Dilation
1	CONV	600×600	8	1	1×1
2	CONV	600×600	8	1	1×1
3	MAX	300×300	8	2	
4	CONV	300×300	16	1	1×1
5	CONV	300×300	16	1	1×1
6	MAX	150×150	16	2	
7	CONV	150×150	96	1	1×1
8	CONV	150×150	96	1	1×1
9	CONV	150×150	96	1	2×1
10	CONV	150×150	96	1	4×2
11	CONV	150×150	96	1	8×4
12	CONV	150×150	96	1	12×8
13	CONV	150×150	96	1	16×12
14	CONV	150×150	96	1	20×16
15	CONV	150×150	96	1	24×20
16	CONV	150×150	96	1	28×24
17	CONV	150×150	96	1	32×28
18	CONV	150×150	96	1	1×32
19	CONV	150×150	16	1	1×1

Table 4.2: Encoder (top part) and context module (bottom part) used as a basis for most networks. All convolutions are 3×3

the number of parameters can become large when coupling image-like feature maps to a fully connected layer, the aim was to create a network similar to that in [6] but reduce the number of parameters. This was done by inserting max pooling layers. In FCNet1 (Figure 4.3), the reduction was performed before the context module, whereas it took place after the context module in FCNet2 (Figure 4.4). The 11 measurements (relative position, steering rate, etc.) from each of the past $n_p = 30$ time steps were flattened to a 330×1 tensor and concatenated with the reduced lidar tensor of the current time step n_c. The network further progressed with three fully connected layers.

4.4.1.2 CNN Only (CNNOnly)

This network, seen in Figure 4.5 relies on convolutions and max poolings, i.e., it has no fully connected layers. Besides running the lidar encoder part, the values in the $n_p = 30$ previous time steps are convolved with a 3×1 kernel over the time dimension. Such convolutions are alternated with max pooling to achieve one scalar per channel in the output of the lidar encoder part. The scalars are then added as a bias before applying the activation function.

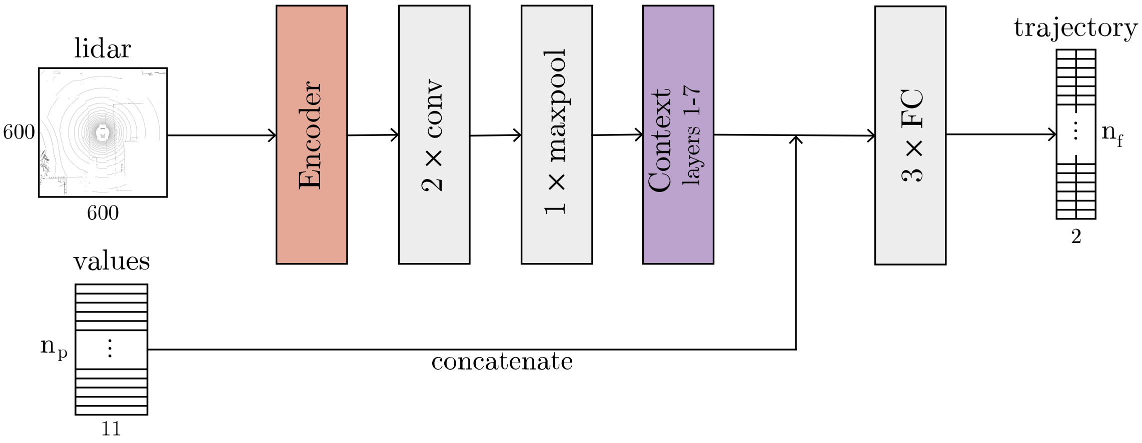


Figure 4.3: Network architecture of FCNet1.

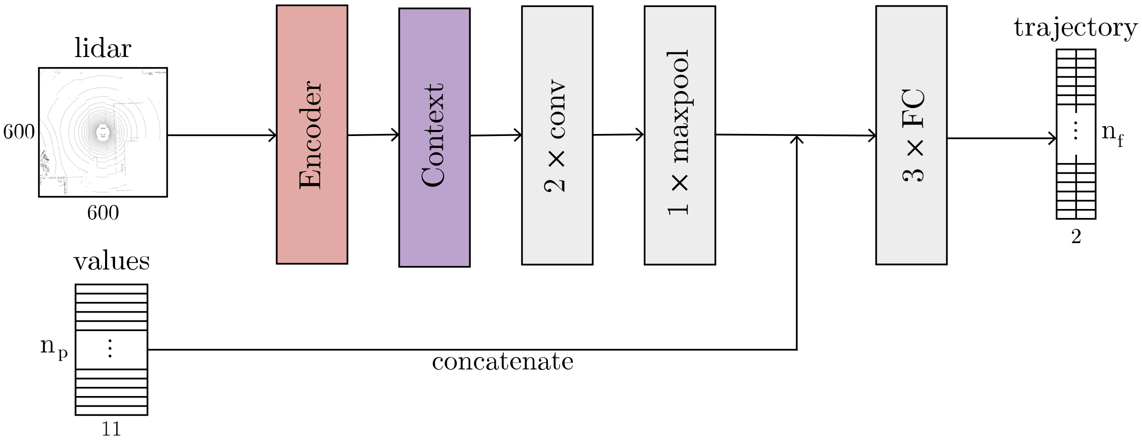


Figure 4.4: Network architecture of FCNet2.

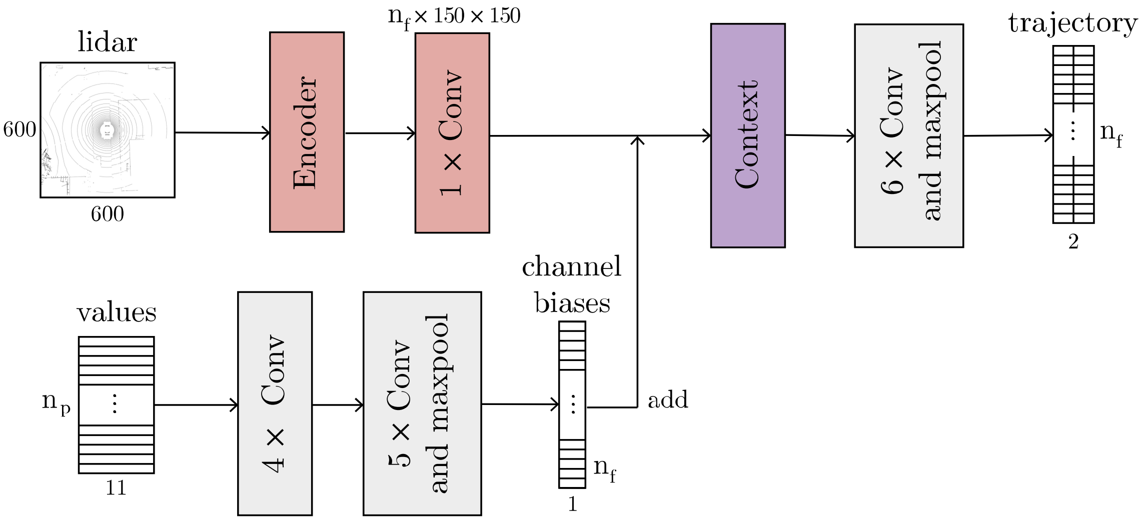


Figure 4.5: Network architecture of CNNOnly.

4.4.1.3 CNN with Bias (CNNBiasFirst, CNNBiasLast, CNNBiasAll)

Another way for the network to utilize the information available from the $n_p = 30$ previous time steps was to incorporate them as a bias in the convolutions. For each kernel in a convolution layer, a scalar value was constructed using a fully connected layer from these measurements to a single output neuron. The bias value corresponding to a particular kernel was added before applying the activation function. An illustration of the concept is shown in Figure 4.6.

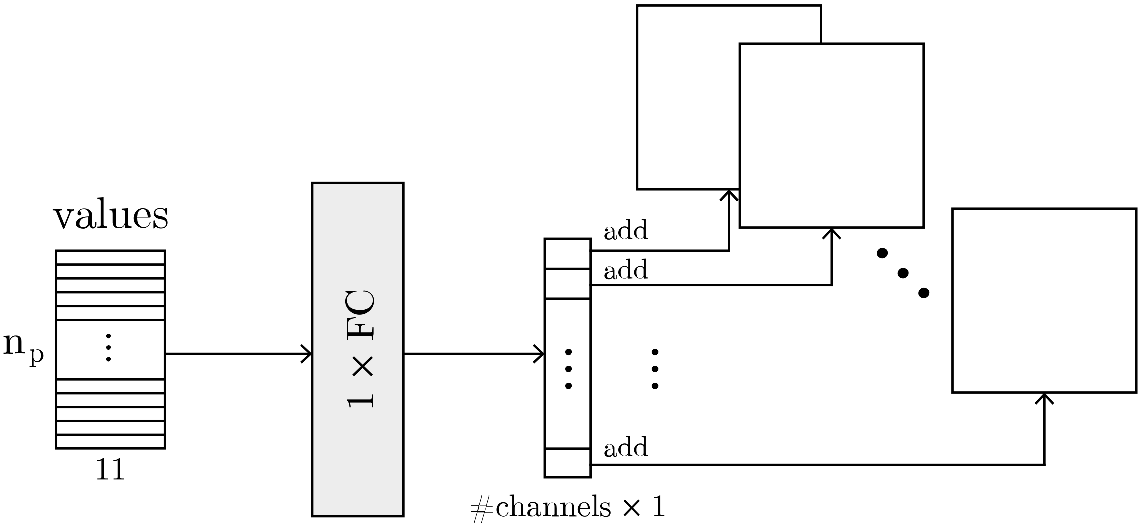


Figure 4.6: Past values form one bias for each channel in a convolution layer.

Three networks were implemented using this technique; one with kernel biases added only in the first layer, one added only in the last layer, and one added to all layers. The final architecture, with bias added to all layers, can be seen in Figure 4.7

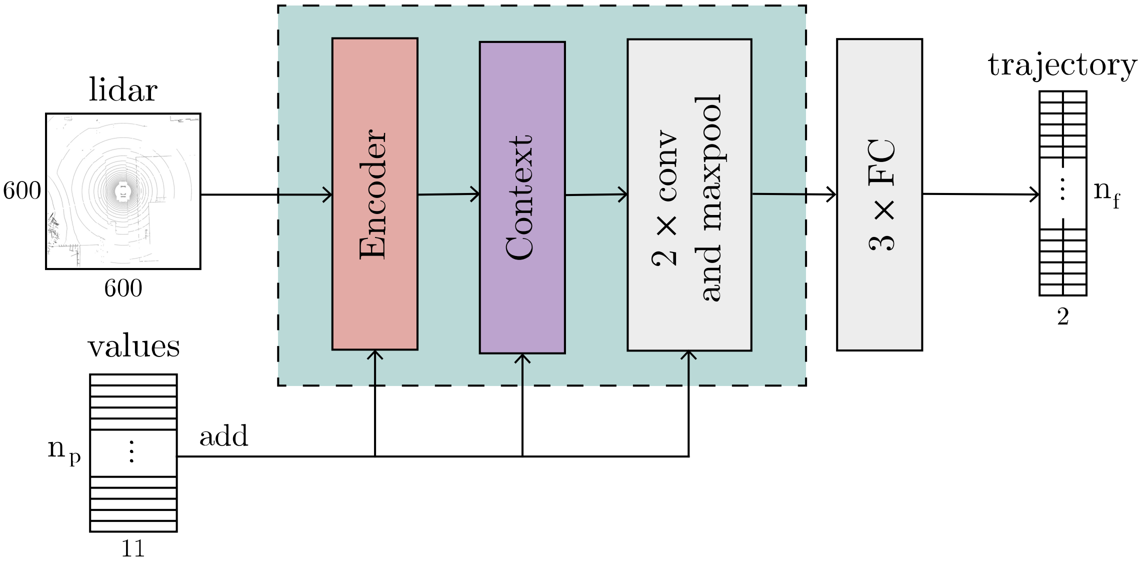


Figure 4.7: Network architecture of CNNBiasAll, applying individual biases to all convolution layers. The PCA, cluster and semantic segmentation idea networks make use of the same structure enclosed in the dashed box, simply referred to as CNNBiasAll.

4.4.1.4 CNN with LSTM (CNNLSTM)

In this network (Figure 4.8), an LSTM network was used to process the measurements from the previous time steps. The output of the LSTM was added as a bias to each kernel in final layer of the encoder part, much like the bias in previously described CNN networks.

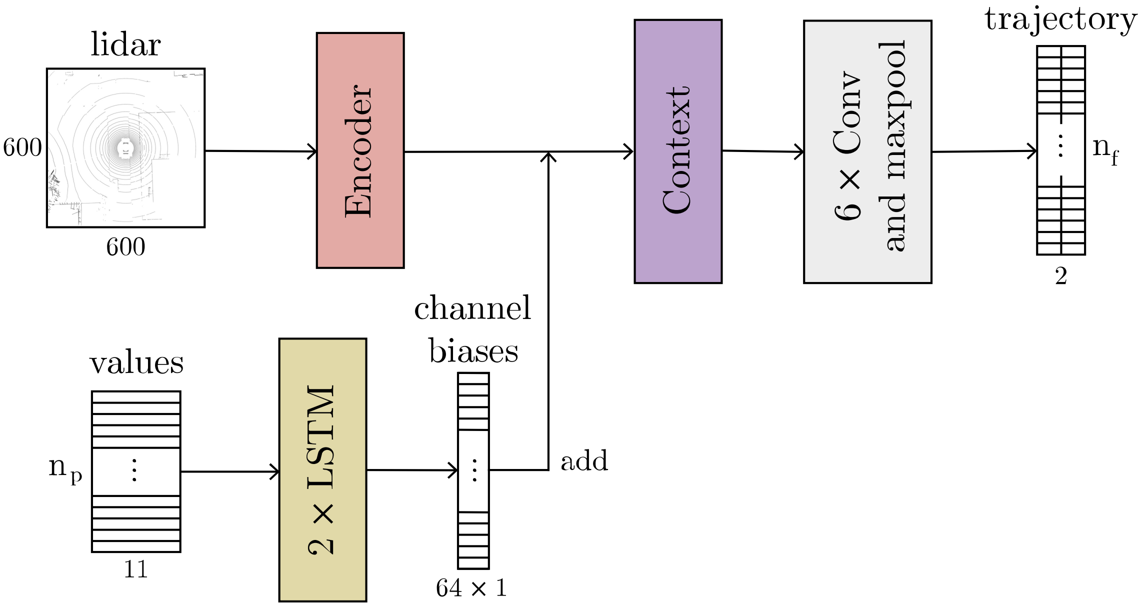


Figure 4.8: Network architecture of CNNwithLSTM.

4.4.1.5 LSTM (LSTMNet)

A network with three LSTM units was constructed with 500 hidden neurons in each. States (lidar and values) from $n_p = 30$ consecutive time steps where included in each iteration of the training, where each input state was used to predict the trajectory in the coming $n_f = 30$ time steps. The architecture is seen in Figure 4.9.

As the states of recurrent units depend on previous states and inputs, it is undesirable to train on non-consecutive data. In other words, the states should contain information about events directly preceding the current input. Therefore, the training was conducted in a consecutive manner, where the input data in two consecutive mini-batches contained data from two consecutive time periods in the recorded data. In other words, the input is timeordered, even between different mini-batches. The hidden state and cell state could therefore be preserved between iterations. As the states are never reset, this way of training makes use of the information available from previous iterations, as would be done at inference time (while driving).

4.4.1.6 LSTM Bidirectional (LSTMNetBi)

This is a bidirectional version of the LSTM network described above, consisting of three units with 500 hidden neurons each. Although the natures of, e.g., text and

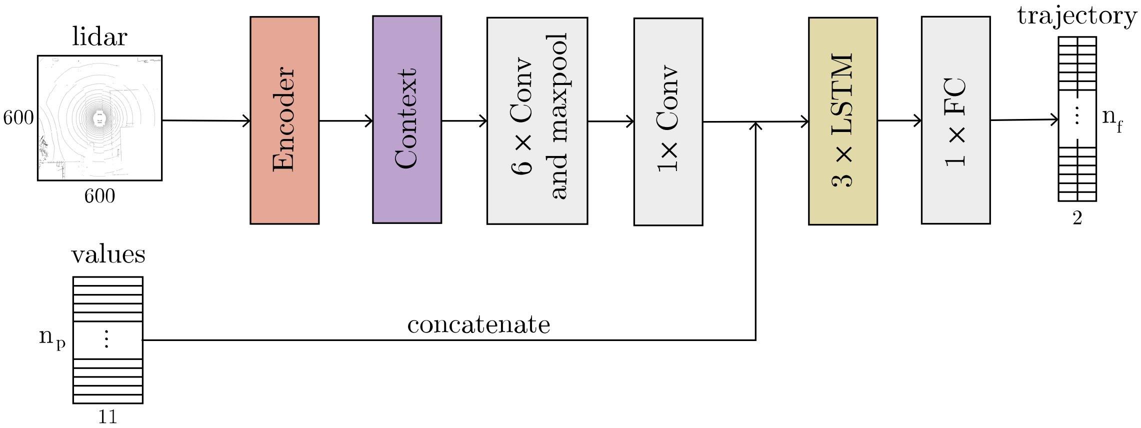


Figure 4.9: Network architecture of LSTMNetwork.

vehicle states are fundamentally different, not least due to the fact that future vehicle states are not available at driving time, the bidirectional approach could be tried at training time. At inference time, both the current state ξ_{n_c} and a previous state $\xi_{n_c-n_p}$ from n_f time steps ago would be part of the input. The LSTM unit should be capable of remembering the input at time $t_c - n_p$, but including state $\xi_{t_c-n_p}$ in the input might act as a reminder. This network is equivalent to the LSTM network but uses bidirectional units instead.

4.4.1.7 GRU (GRUNet)

This network is equivalent to the LSTM network but uses GRU units instead.

4.4.2 PCA Regression Networks

The PCA regression network predicts some of the leading principal components of the trajectories recorded, as explained in 3.3.2. It is similar in architecture to the pure regression networks, with the exception of the last part, which transforms the estimated principal component values into a full trajectory with coordinates for the $n_f = 30$ future time steps. The resulting trajectory is then compared with the ground truth using MSE. The PCA regression Network can be seen in Figure 4.10.

4.4.3 Clustering Networks

The clustering network predicts which class the future trajectory belongs to, as well as adjustments to the principal components of the class-typical trajectory. The output of the classification part is a vector, seen as a probability distribution over the possible classes. Since the ground truth is known, the distribution is a vector of all zeros, except the element representing the correct class which is equal to one, forming a so called one-hot vector. The classification loss is then calculated using the cross-entropy between the class prediction distribution and the ground truth distribution. In Figure 4.11 a schematic figure of the architecture is shown. Three versions where trained, using 10, 20 and 100 clusters respectively.

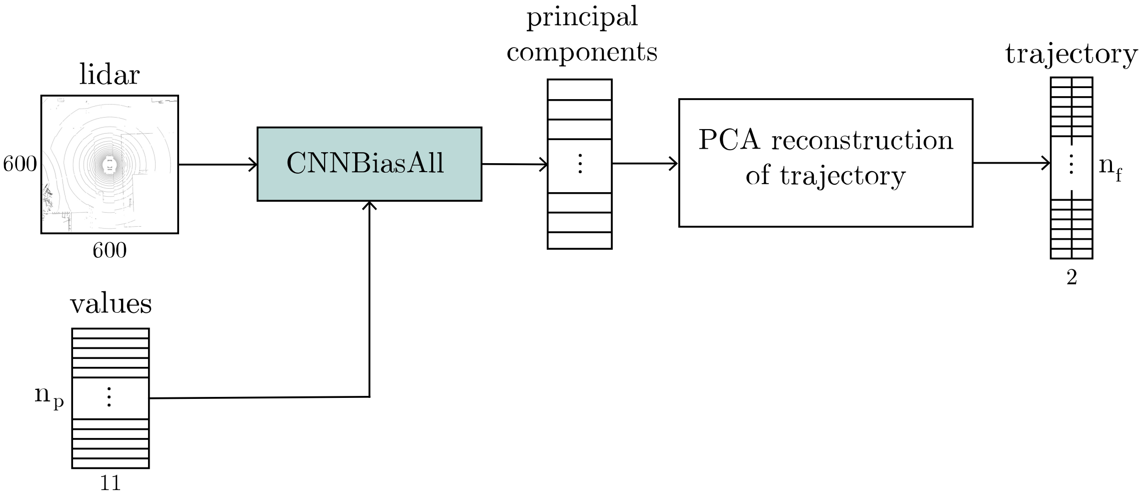


Figure 4.10: General architecture of the PCA idea networks.

As mentioned, the principal component part predicts *adjustments* to seven of the leading principal components of each possible class. The adjustments are then added to the principal components of the predicted class. The predicted class is taken to be the one with highest probability in the predicted class distribution, although it might not be the ground truth class. This is in hope that the classification accuracy will increase to an acceptable level and that the predicted adjustments will bring some robustness to misclassification. The MSE is calculated on the adjusted trajectory, which is added to the cross-entropy loss of the classification. Added together, the two losses form the total loss which is the starting point of the backpropagation.

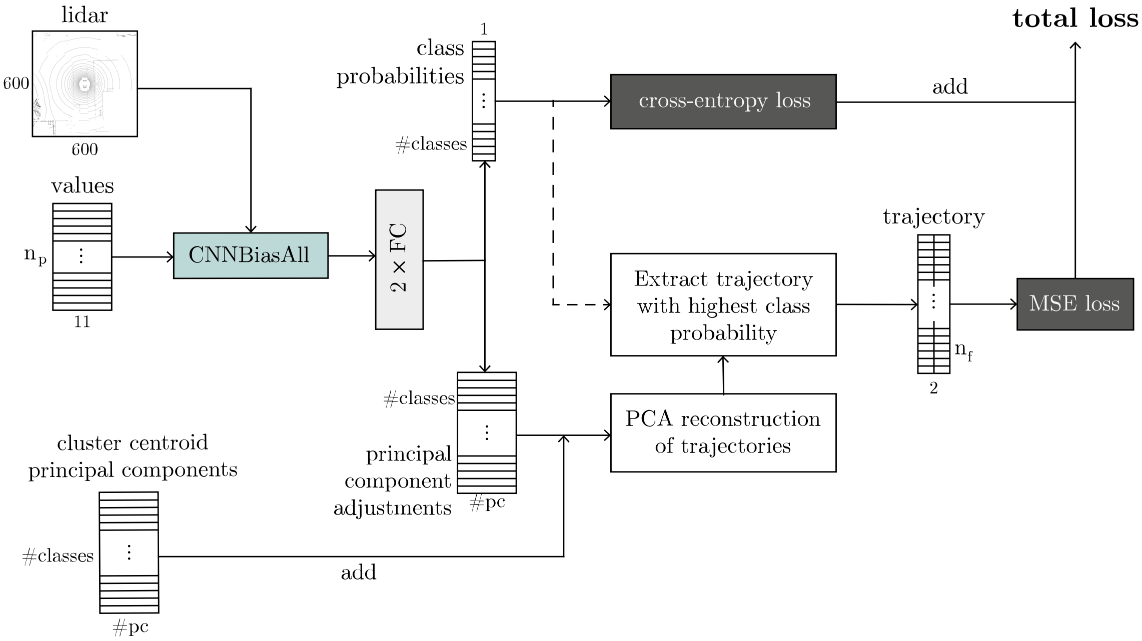


Figure 4.11: Architecture of the cluster idea network. The dashed line indicates how the trajectory with highest class probability is extracted using information from the class probabilities, but that backpropagation is not applicable along that line.

4.4.4 Semantic Segmentation Networks

The semantic segmentation network (Figure 4.12) is basically the CNNBiasAll network without any further processing after the context module, although with different number of channels in the last layer. It gives as output one semantic map per future predicted time step, where each pixel value in a feature map indicates a scaled probability of being occupied by the vehicle at the time step corresponding to that semantic map. Note that each pixel value corresponds to the output of the logistic sigmoid function (in $[0, 1]$), and that the binary cross-entropy is used to classify each individual pixel. This means that the values in a feature map does in general not sum to one and is thus not strictly speaking a probability distribution. However, scaling the pixel values would easily yield a valid probability distribution, and it can therefore be thought of as one for simplicity.

The center of mass of a semantic map is calculated to form the predicted coordinates. Note that the MSE is merely used to compare the loss with that of other models, whereas the binary cross-entropy is used for training. As the cross-entropy loss becomes smaller, so does the MSE loss. In contrast, training on the MSE loss does not ensure the pixel values converge towards the ground truth semantic maps; the center of mass can be equal to the ground truth coordinates for many semantic maps different from the ground truth.

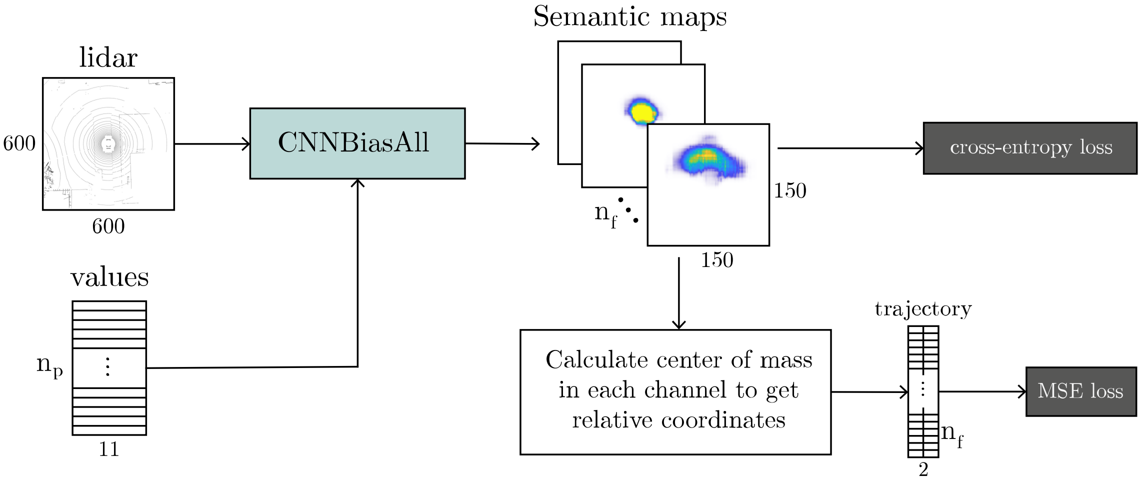


Figure 4.12: Architecture of the semantic segmentation network. The network is trained using the binary cross-entropy loss, whereas the MSE loss is used merely to compare the model with others.

4.5 Training

The training procedure changed somewhat during the course of the project. This is a result of learning how Pytorch can best be utilized, that the mini-batch SGD and Adam optimization techniques yield substantially different results, and realizing just how time consuming neural network training can be.

In the beginning of the project, referred to as the *experimentation phase*, the mini-batch size for each of the non-recurrent models was set to 16. This seemed to be the largest value which could be used among all networks without running out of memory on the GPU. For the recurrent models, which required reading data in a continuous fashion, the mini-batch size was set to one (i.e., common stochastic gradient descent) simply to reduce the implementation complexity.

These first networks trained during the experimentation phase (see Figure 5.1) were trained using Mini-batch SGD with momentum, in conjunction with two separate schedulers for learning rate and momentum. The scheduler continuously increases the learning rate for one epoch (i.e., one pass through the training data) up to a maximum value, after which it decreases back to its initial value for another epoch. The reason for using this kind of scheduling is to first let the network slowly ”warm up”, i.e., set the parameter values to something better than random, then to take as large steps as possible for a quicker convergence, and then to slowly settle and allow for some fine tuning. The part where large learning rates are applied has some potentially regularizing effects, as it allows for escaping local minima encountered during the warm up phase [36]. The maximum learning rate values were found using a so called *learning rate finder* algorithm, and ranged between 10^{-6} and $3 \cdot 10^{-4}$ for the different networks. The goal of the learning rate finder is to find the largest learning rate value which does not cause the loss to diverge, thus speeding up training. It starts by training a network with a given optimization algorithm, in this case Mini-batch SGD, using a very small learning rate, e.g., 10^{-8}. The learning rate is then continuously increased during the training and will eventually be large enough to cause the (smoothed) loss to diverge. The learning rate is then chosen to be about one order of magnitude smaller than the value causing loss divergence, simply to not risk diverging [37].

The latter networks were trained using the Adam optimization technique, as it yielded better performance. The learning rate was set to 10^{-5} and a weight decay value was set to 10^{-5} for Adam. Note that Adam makes use of adaptive learning rate and that using a scheduler in addition would thwart the benefits.

Because some of the networks trained later were larger, i.e., had more parameters, a mini-batch size of 12 was chosen instead of 16 as to fit on the GPU. Moreover, these were trained using a time limit of seven hours instead of training for two epochs. The idea is that although some networks might yield good results, their training time might be substantially longer than others and therefore not feasible to train until convergence.

4.6 Control Technique

The model predictive controller was implemented by Juliano Pinto and Dapeng Liu, both of whom are PhD students at the department of Electrical Engineering, Chalmers University of Technology. For each time step t the MPC is fed the predicted trajectory $\hat{p}$ consisting of *global* coordinates for n_f future time steps, and the current state of the car (global coordinates, acceleration, velocity, yaw). The MPC then optimizes the steering signals (i.e., throttle, brake, steering angle) to best follow the desired trajectory, while also not breaking any constraints. Environmental constraints, used to prevent going off road or hitting an object, is based on the distance maps described in Section 2.8.1. The MPC must be able to locate the car within the distance maps and thus the coordinates must be global, as mentioned before. The MPC must also hold the physical constraints of the car, e.g., acceleration or turn ratio.

The MPC optimization algorithm is dependent on an accurate model of the vehicle. However the car simulation in CARLA is based on NVIDIA's PhysX engine, which makes the car model very realistic but also very difficult to accurately recreate in the MPC. Especially, the relationship between the acceleration of the car and the steering signals throttle and brake was difficult to model, which is why a neural network was used to approximate the relationship. In order to facilitate for the network, the car simulator was restricted to only have one gear (and reverse), which covered a large velocity range of about $0 - 100$ km/h. This is somewhat unrealistic but it greatly simplified the complexity of the acceleration-steering signal relationship and also saved time. The neural network was trained on 30 min of data and an example of the performance can be seen in Figure 4.13.

The level of precision can be adjusted, but as a trade off to computing time. As the connection between the server (CARLA simulator) and the client (the module controlling the car) can time out, the computation time must be kept low. The computation time depends on the number of future coordinates $m_f \in (\mathbb{Z}|m_f \leq n_f)$ the MPC needs to regard, and also the maximum number of optimization steps allowed. By using every third of the n_f future coordinates (i.e., $m_f = n_f/3$) and setting the maximum optimization steps to 300 was enough for most situations.

The MPC might not always find the optimal route as it has a time constraint, and sometimes is cannot find any feasible path. Furthermore the MPC returns the steering signals for m_f time steps, but only the steering signals for the next time step $t + 1$ is forwarded to the car simulator, which performs the action.

4.7 Network evaluation

The networks will be evaluated by comparing the results with a baseline and also through visual inspection. A *baseline* is the average loss of the output from a (typically) simple and conventional algorithm. The baseline for our problem, predicting

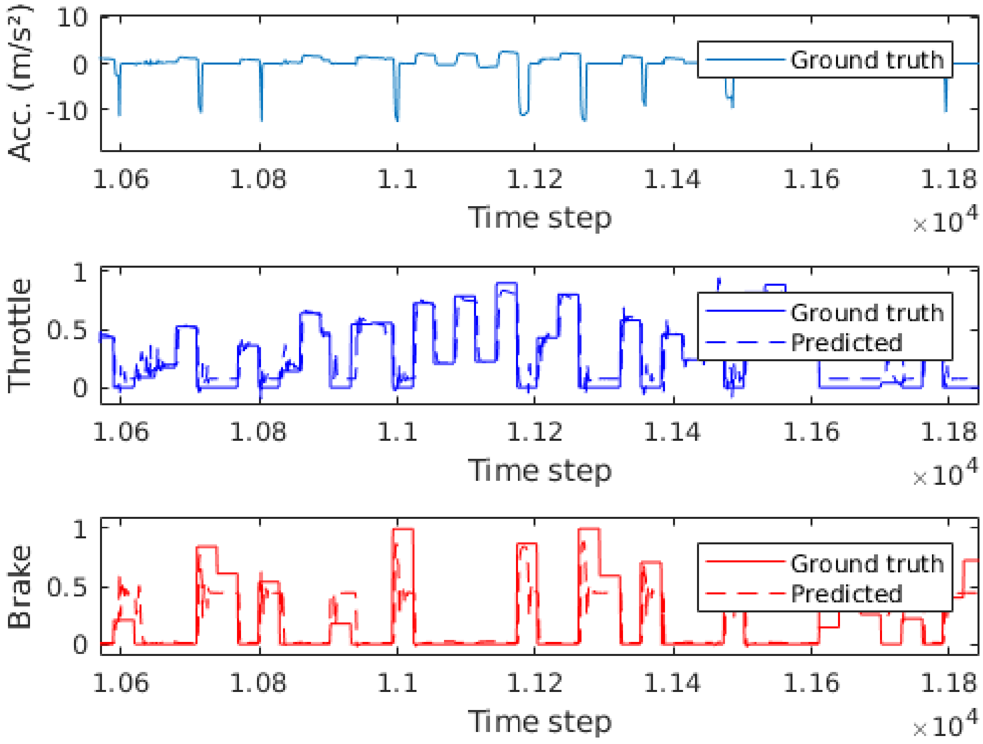


Figure 4.13: Top: Acceleration over time as an input to the dynamics approximation network. Middle: Predicted and ground truth throttle values for the corresponding acceleration in the top plot. Bottom: Predicted and ground truth brake signals. It can be seen that the predicted signals roughly follow the ground truth. For example, predicted brake signals are applied when a deceleration is present.

a driving trajectory, is created from the future locations of the car without changing the direction or velocity. In other words, given the state of the car at t the next coordinates at $t+1$ are calculated by

$$[x_{t+1}, y_{t+1}] = [x_t, y_t] + v_t * 0.1, \tag{4.1}$$

where 0.1 seconds is the defined length of a time step. The loss is derived and the procedure is repeated for all data points in the data set. The average loss is the baseline of the data set, see Table 4.3 for the respective baselines used. To conclude, the networks should perform *at least* as good as the baseline to add any value.

Another manner of evaluating the networks was through visual inspection. By plotting the output trajectory from the network against the ground truth upon the topview lidar image, one can gauge its performance. It is important to keep in mind that the output trajectory must not exactly match the ground truth, as driving a car is not deterministic. However the output trajectory must look realistic, consequently it must follow the road, stop for obstacles and not crash.

Data set	**Baseline**
Train	6.894
Validation	7.670
Test	7.591

Table 4.3: The baseline for respective data set.

5
Results

After implementing and training the various networks described in the method section (Section 4), the results were evaluated using two measures, namely the loss of the predicted trajectory and visual inspection, both of which are presented below. The results are presented respectively for each of the four main ideas described in Section 3.3.

5.1 Network Loss

The loss is a quantitative measure of how close the predicted trajectories are to the ground truth trajectories. Even though some networks were trained using different losses, the evaluation is done only on the MSE, as it enables fair comparison between networks.

5.1.1 Experimentation Phase

In the experimentation phase, ten different network architectures were tried in search for the best overall pure regression architecture. They were trained for two epochs each, but it should be noted that the expenditure of training time differed vastly among the networks. As can be seen in Figure 5.1, some networks achieved approximately the same performance as the baseline, while others were significantly poorer. Table **??** presents the minimum validation loss achieved on each of the architectures.

The recurrent networks named LSTMNet, LSTMNetBi and GRUNet seemed to converge towards the same loss value, although very slowly. As the training of these recurrent networks was quite different compared to the rest, i.e., not balancing data nor shuffling it, a totally fair comparison cannot be made. However, as they took longer time to train and obtained worse performance than the non-recurrent networks, they were not explored further. When observing Figure 5.1, it can be seen that the recurrent networks have been trained for approximately half as many iterations as the rest of the networks. This is an artifact of reading 30 time steps in each mini-batch, using a mini-batch size of one. Thus, the recurrent networks processed the same amount of data in approximately half the number of iterations compared to using a mini-batch size of 16 as the other networks.

The networks called CNNLSTM and CNNOnly were considerably worse than the other networks. While CNNLSTM did not seem to learn anything of value, CN-

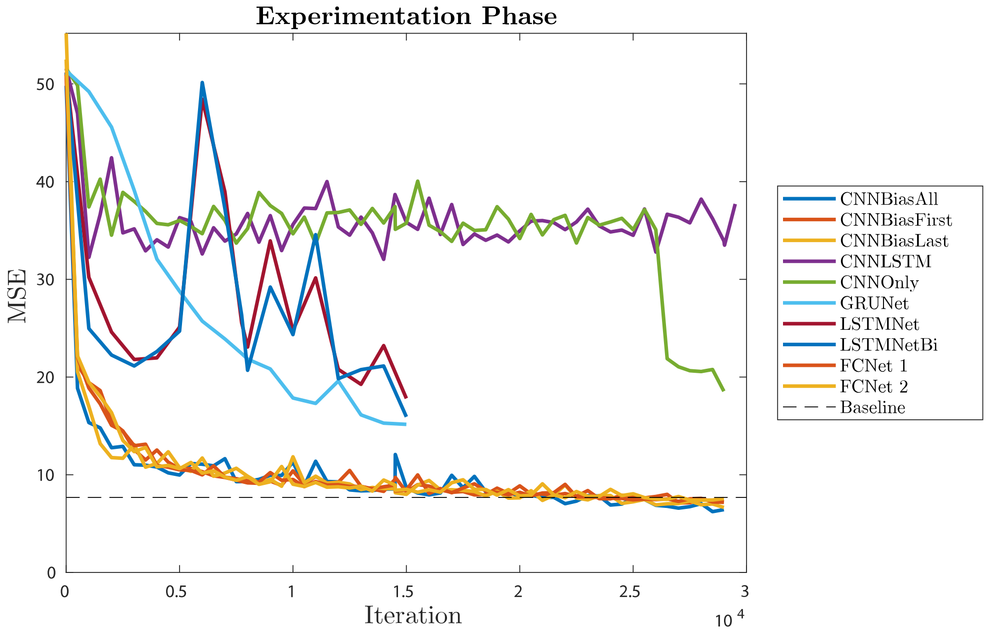


Figure 5.1: Validation loss for ten first pure regression network architechtures implemented. It can be seen that the networks called CNNBiasFirst, CNNBiasLast and CNNBiasAll have better performance than the rest.

Network	Minimum validation loss
CNNBiasAll	**6.2607**
CNNBiasFirst	7.1126
CNNBiasLast	6.6577
CNNLSTM	32.0375
CNNOnly	18.5160
GRUNet	15.1629
LSTMNet	17.8025
LSTMNetBi	15.9195
FCNet 1	7.1495
FCNet 2	7.4194

Table 5.1: Minimum validation loss obtained for different network architectures (trained using the mini-batch SGD optimization technique).

NOnly appears to have ”realized” something important towards the final iterations (i.e., escaped a local minimum), almost instantaneously dropping in loss.

In conclusion, the recurrent networks did not seem very promising. Among the networks with better performance, there was little difference. However, CNNBiasAll seemed slightly better. For this reason, the same architecture was trained another time using the Adam optimization algorithm. A comparison of the loss can be seen in Figure 5.2 and confirms the findings in [15], i.e., Adam is the superior optimization method. Thus it was used in all future training.

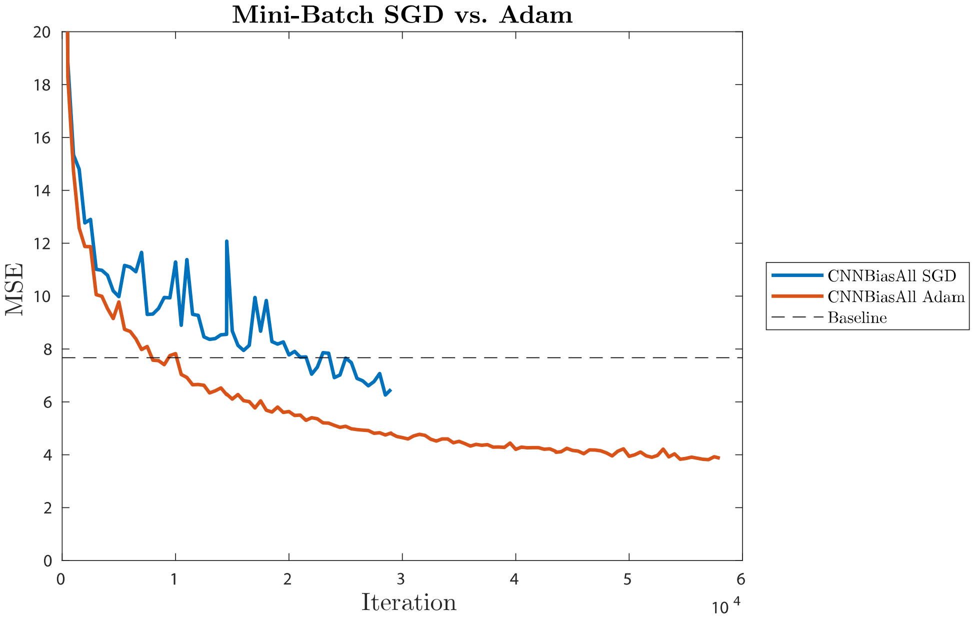


Figure 5.2: When training the same network architecture, Adam is superior to Mini-batch SGD, as the validation loss decreases more quickly and ultimately reaches lower values. Note that the network trained using Mini-batch SGD did not train for as many iterations, but the trend is still clearly visible.

5.1.2 Pure Regression Idea

The CNNBiasAll architecture was tested with different number of layers, number of channels in the convolutions, number of maxpoolings, different dilation sizes and different convolution strides. For a more detailed specification of the different networks, see our github repository[1].

The validation losses of each architecture, within the pure regression idea, are plotted over the number of training iterations in Figure 5.3. It illustrates how similar the performance was between different architectures. The network *More Layers 2* seemed the most promising, as it yielded the lowest test loss (see Table **??**), and the training was therefore continued for another 7 hours. Unfortunately, there was not much difference after the additional 7 hours. The network *More layers 1* was also trained for a total of 14 hours, obtaining similar but slightly poorer performance.
Since the network should be able to slow down for cars in front of the vehicle, it must have some information about the movement of other objects. As loading just one extra lidar frame form the previous time step resulted in a doubling in training time, see Figure 5.4, and because the results were no better than using a single lidar frame, it was decided to train using only one frame.

[1] `https://github.com/AnnaNylander/exjobb/tree/master/network/architectures`

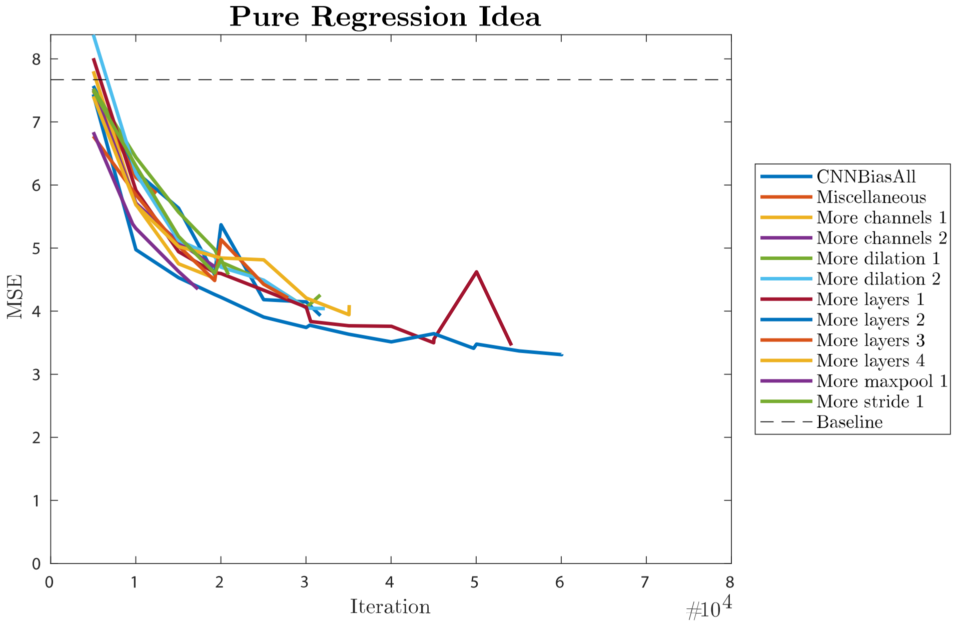


Figure 5.3: Validation loss for the pure regression idea networks. The networks named *More layers 1* and *More layers 2* were trained for 14 hours, whereas the rest were trained for 7 hours each.

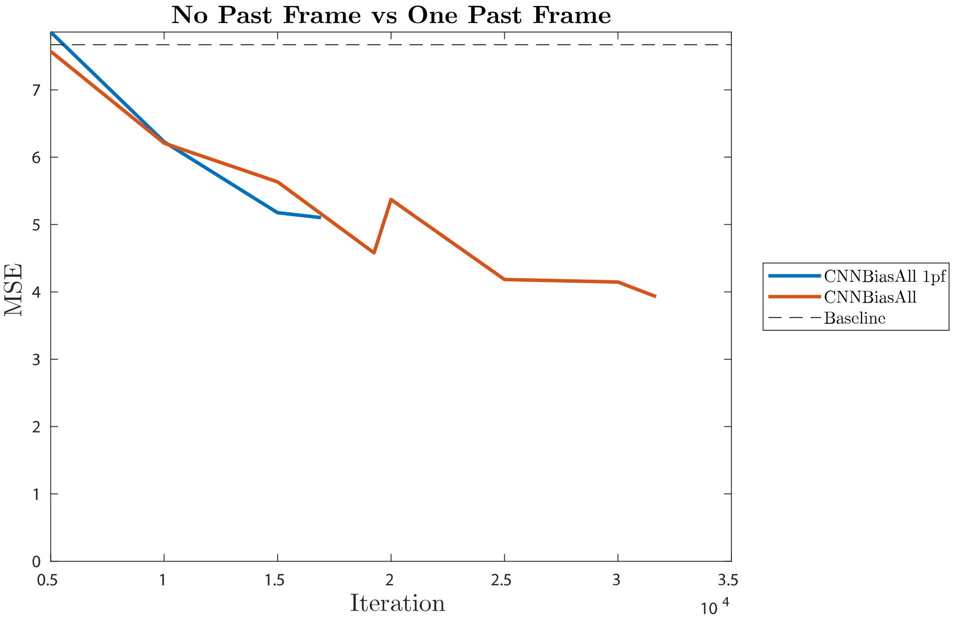


Figure 5.4: Validation loss for the *CNNBiasAll* network using only the current lidar frame and using the current and previous lidar frames. No effective gain in performance was observed, whereas the training speed was halved.

Network	**Test loss (7h)**	**Test loss (14h)**
Baseline	7.591	7.591
CNNBiasAll	3.8872	
Miscellaneous 1	5.6770	
More channels 1	4.4894	
More channels 2	4.5699	
More dilation 1	3.9167	
More layers 1	3.8987	3.5178
More layers 2	**3.6866**	**3.2072**
More layers 3	3.9121	
More layers 4	3.9933	
More maxpool 1	4.2930	
More stride	4.5891	

Table 5.2: Test loss obtained for the pure regression idea networks. Training *More layers 2* for a total of 14 hours yielded a small gain in performance, compared to training for only 7 hours.

5.1.3 PCA Idea

As can be seen in Figure 5.5, there is little difference between using 5 or 20 principal components. The PCA idea seems to perform approximately as good as the pure regression idea. The network with lowest test loss after seven hours of training, namely the network using 20 components, was trained for a total of 14 hours. However, as presented in Section 5.2, the trajectories became less smooth when using more principal components, indicating that 20 dimensions perhaps was too many. For this reason, the network using 5 principal components was trained equally long, reaching approximately the same test loss (see Table **??**) while maintaining the smoothness of the trajectories.

Network	**Test loss (7h)**	**Test loss (14h)**
Baseline	7.591	7.591
5 components	4.5116	3.8046
7 components	4.5667	
10 components	4.4443	
20 components	**4.3247**	**3.6317**

Table 5.3: Test loss obtained for the PCA idea networks.

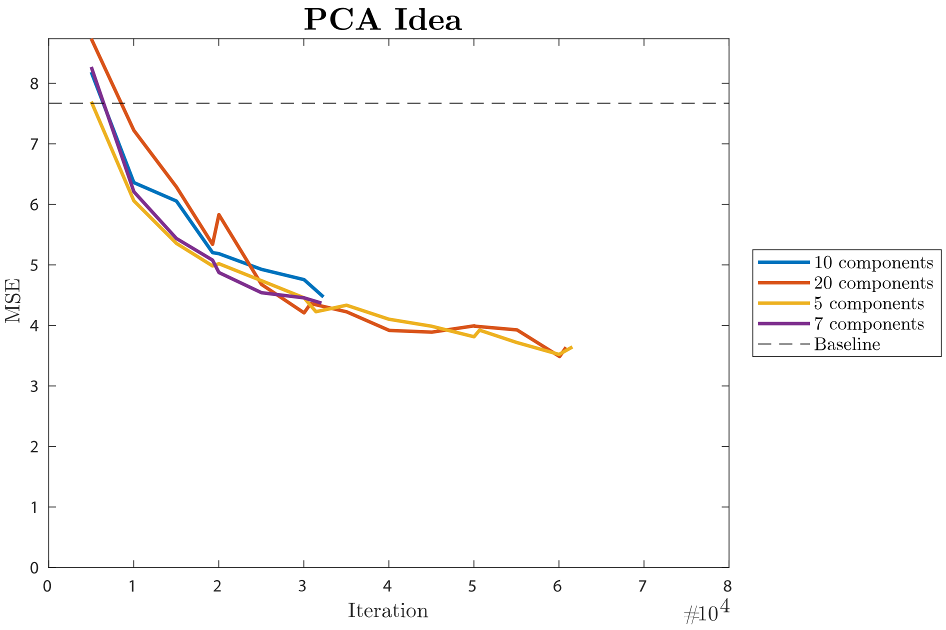


Figure 5.5: Validation loss for the PCA idea networks. The difference in performance was similar when using 5, 7, 10 and 20 components respectively.

5.1.4 Cluster Idea

The cluster idea yielded somewhat larger test losses compared to the PCA and pure regression ideas, see Table **??**. The number of clusters does not seem to significantly affect the performance, at least not within the limited training time. The network with 20 clusters yielded lowest validation loss during the first seven hours, which was why it was trained for another 14 hours. As can be seen in Figure 5.6, the network using 100 clusters was trained further as well, out of curiosity. The same figure and Table **??** indicate that too little training was not the issue, but rather the relatively high number of clusters.

Network	**Test loss (7h)**	**Test loss (14h)**
Baseline	7.591	7.591
10 clusters	6.0589	
20 clusters	**5.6275**	**4.5555**
100 clusters	5.9859	4.9808

Table 5.4: Test loss obtained for the cluster idea networks.

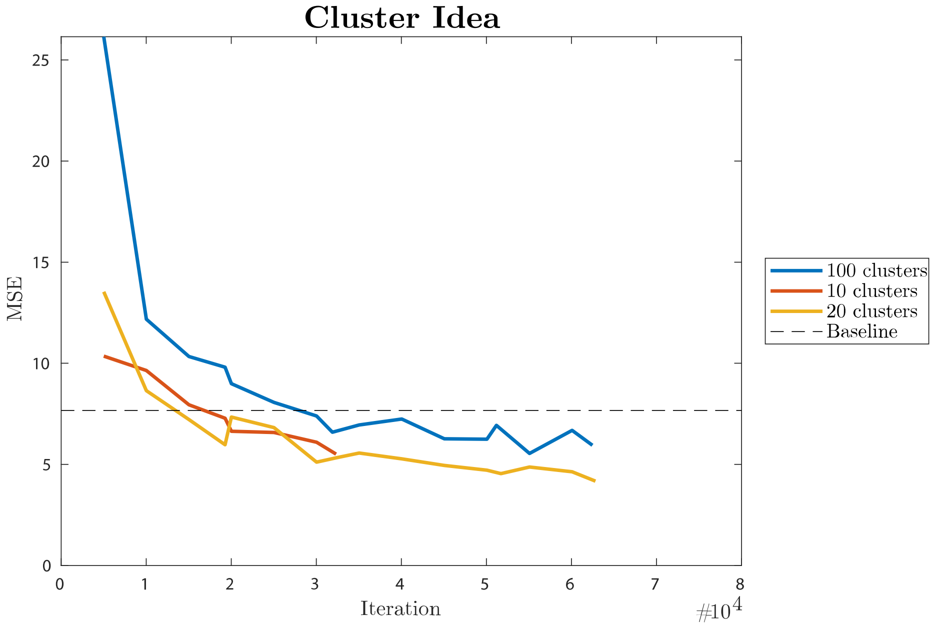


Figure 5.6: Validation loss for the cluster idea networks.

5.1.5 Semantic Segmentation Idea

The semantic segmentation idea seems to yield considerably higher MSE loss compared to the rest of the ideas, both on the validation (see Figure 5.7) and test set (see Table **??**). It is the only network idea which did not reach the baseline performance. The three networks were trained using different radii of the location in the ground truth semantic map. The network labelled *Radius 6-5-4-3* started training using a radius of 6 m for three hours, then 5 m for two hours, 4 m for one hour and finally 3 m for one hour. The training procedure was similar for the network labelled *Radius 10-8-6-4*, but which was trained another seven hours using a radius of 4 m. The *Radius 4* network was trained using a radius of 4 m all seven hours.

Network	**Test loss (7h)**	**Test loss (14h)**
Baseline	**7.591**	**7.591**
Radius 10-8-6-4	9.8988	8.6059
Radius 4	15.2244	
Radius 6-5-4-3	12.3799	

Table 5.5: Test loss obtained for the semantic segmentation idea networks. Using a larger radius seems to decrease the loss more quickly. The network *Radius 10-8-6-4* was trained for 14 hours but did not reach the baseline loss.

Finally, when comparing the validation losses in Figure 5.8 for the networks with lowest test loss in each main idea, it is evident that all ideas except for the semantic segmentation idea are approximately equal in performance. The network with smallest loss on the test set after training 14 hours, namely *More layers 2* with a test loss of 3.2072, was further trained until it reached a total of 63 hours. The test loss

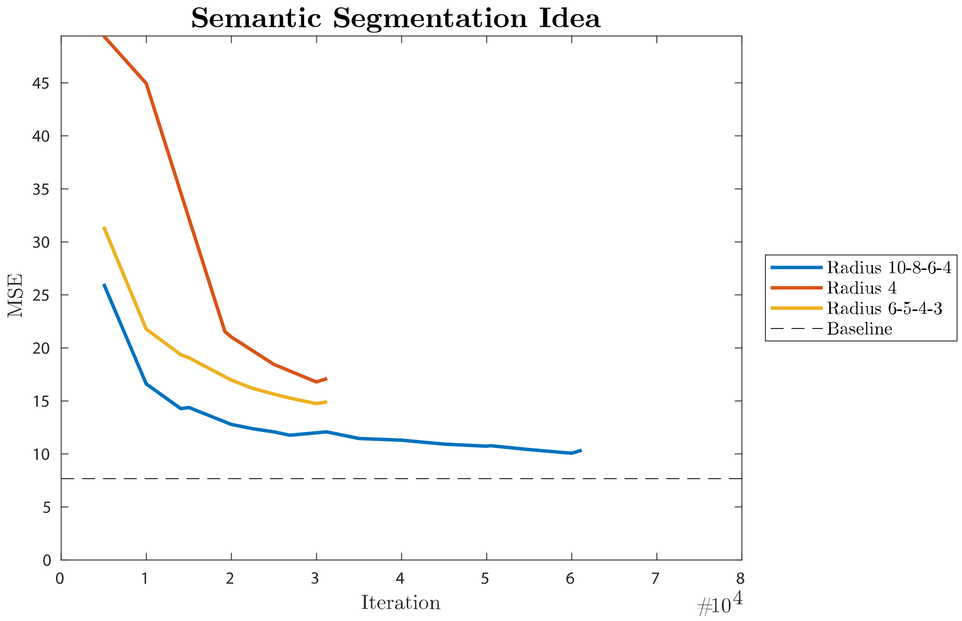


Figure 5.7: Validation loss for the semantic segmentation idea networks.

obtained was 2.5922, which is significantly smaller than after 14 hours of training, although the difference is quite small with respect to the additional training time. In the visual inspection section, the model trained for 14 hours will be used for a fair comparison with the rest of the networks.

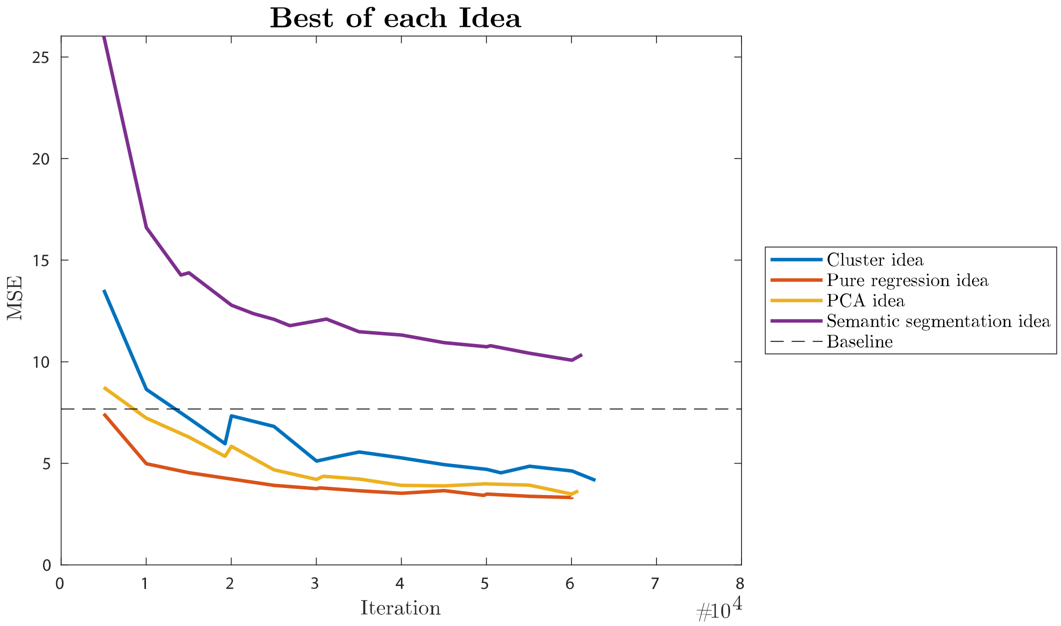


Figure 5.8: Validation loss for the networks with lowest test loss for each of the main ideas. There is little difference in performance between the ideas, except for the semantic segmentation idea which performs worse.

5.2 Visual Inspection

The predicted trajectories of the best network for each idea, i.e., having lowest loss on the test set, were plotted in blue on top of the top view lidar image, together with the past (red) and the future ground truth (green) trajectories. An exception to this definition of the best network is for the PCA idea network. Using 5 and 20 principal components respectively did not make a large difference on the test loss, but the network using 20 principal components had significantly more ”squiggly” trajectory predictions. As smoother trajectory predictions was a motivation behind the PCA idea, the network using 5 principal components was deemed the best of the PCA idea networks, although yielding slightly higher test loss. The following text is an interpretation of the plots, describing the results in different traffic situations. Note that the figures have been selected to showcase both good and bad predictions in these situations. In two of the presented situations, the network *More layers 1* was used instead of *More layers 2*, even though *More layers 2* yielded lower test loss. This was done in order to more clearly demonstrate the behaviour in the situation at hand.

Figure 5.9 shows the behaviour in a left turn without intention, meaning that the car has no choice but to follow the road to the left. This is one of the most basic scenarios, since there is no choice to be made regarding the direction, no traffic lights to follow and no pedestrians or cars that can (in a legal way) cross the road at this point. It is evident from the top row of Figure 5.9 that the pure regression and cluster idea models are able to produce reasonable trajectories. However, the bottom row of the same figure depicts a similar left turn where it can be seen that the predicted trajectories continue outside the road. Note how the trajectories of the cluster idea model and PCA idea model are similar to those of the pure regression model, while being smoother and more realistic.

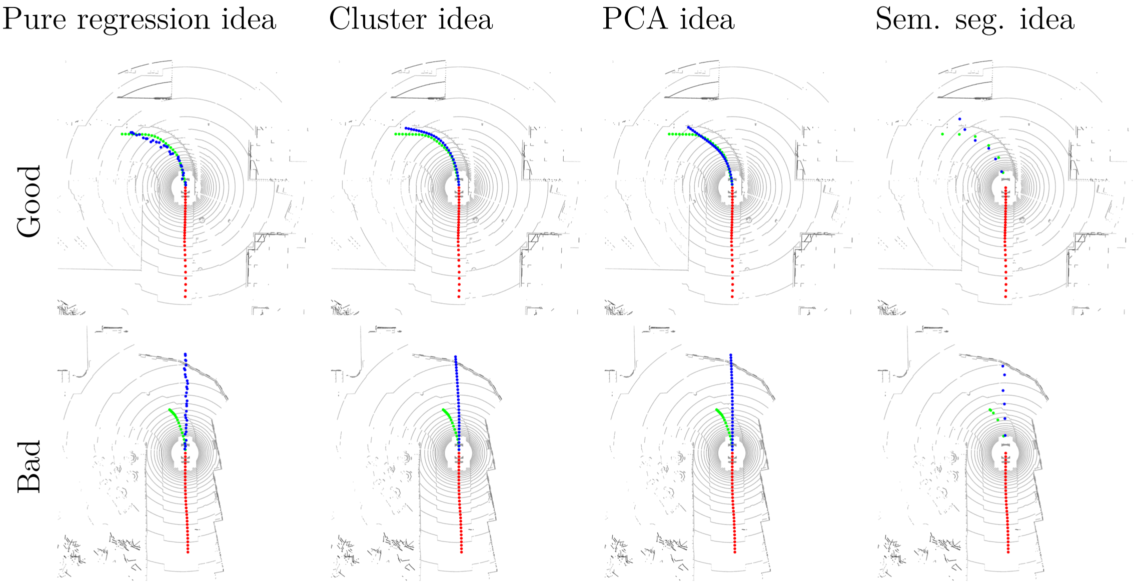


Figure 5.9: Turning left *without* intention. Top: Pure regression and cluster idea models seem similarly accurate, while PCA idea and semantic segmentation idea networks struggle. Bottom: another left turn which requires the network to follow to road. It can be seen in the lidar images that a left turn is crucial, but all four models struggle to predict an acceptable trajectory.

Figure 5.10 shows a crossroads where the car is directed to take a left turn. As can be seen from the top row, pure regression, cluster and PCA idea models manage this particular situation quite well, while the semantic segmentation idea model struggles again. The bottom row displays the same turn as the top row but at a later time step, where the car is approximately half way through the turn. All four models tend to undershoot with respect to the ground truth trajectory.

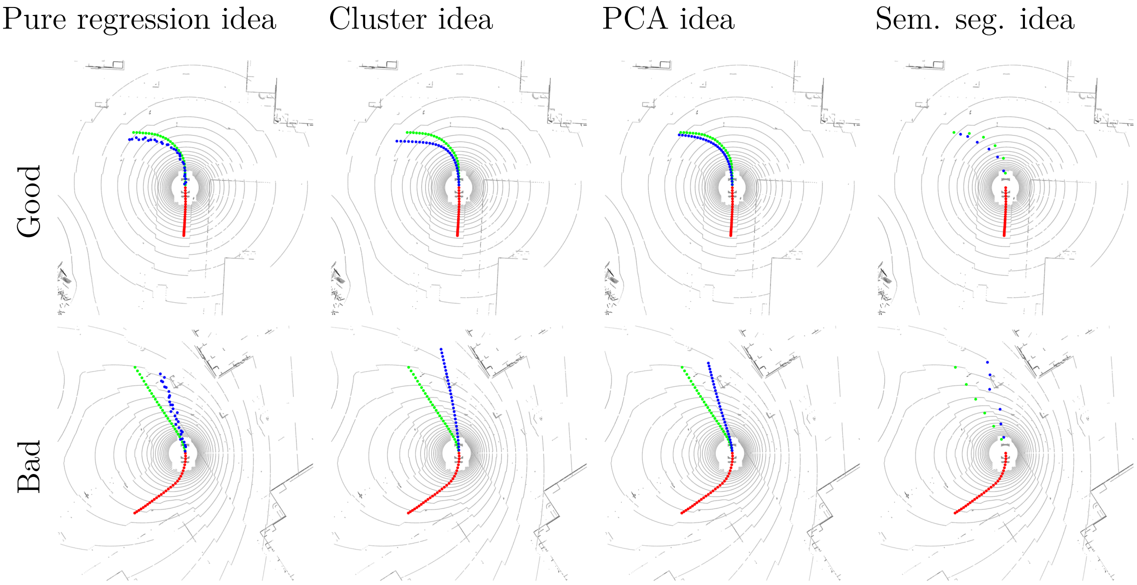


Figure 5.10: Turning left *with* intention (i.e., instructed to turn left in a crossroads).Top: Most models predict fairly accurate trajectories. Bottom: All models undershoot with respect to the ground truth trajectory.

An even simpler situation than the left turn without intention would be to follow a straight road. The top row of Figure 5.11 shows how the four models manage to predict trajectories that align very well with the ground truth. The bottom row presents a situation where the car is driving in a zigzag fashion due to the added steering noise. All four models fail to predict trajectories that correct for the perturbation, and continue in the current direction. Note that although the direction is not desirable, the speed (spacing between points) is quite accurate. It is also worth noting that the semantic segmentation network predicts four points in a cluster towards the top of the image, indicating it has not learned to predict speed as good as the other networks.

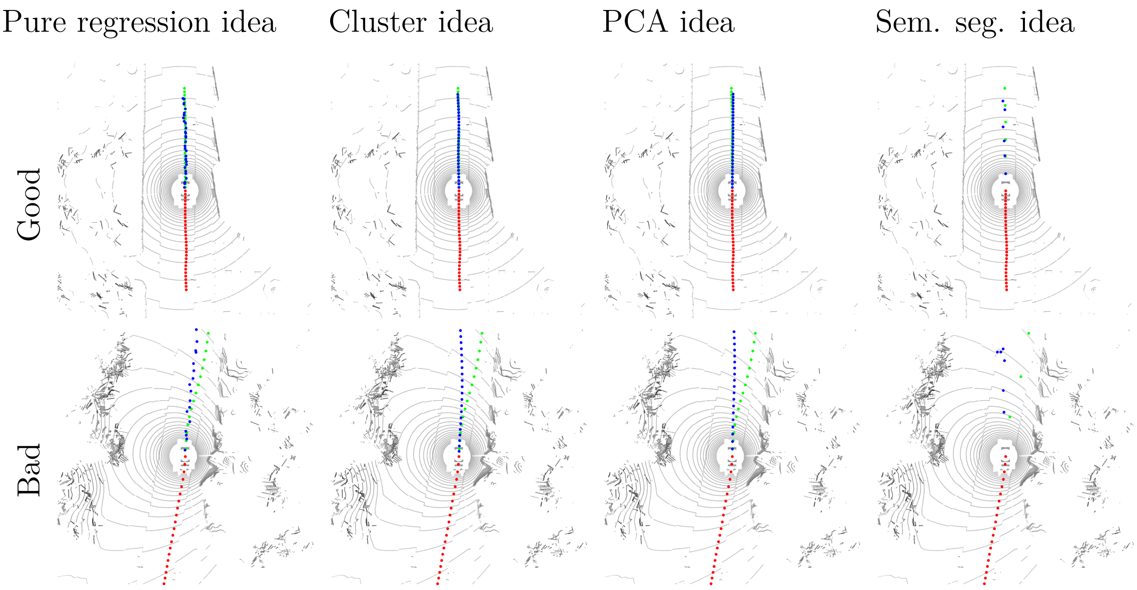


Figure 5.11: Going straight ahead. Top: All four models are accurate in this rather simple situation. Bottom: All four models fail to correct for the perturbation of the steering signal.

In both rows of Figure 5.12 the car can be seen waiting for a traffic light to turn green. In the top row, the pure regression, cluster and PCA networks seem to handle the situation correctly, while the semantic segmentation network predicts some small movement. However, the predictions on the bottom row show that the pure regression and PCA idea networks would like to go past the traffic light, despite being red. The cluster and semantic segmentation networks show similar behaviour, but to a lesser degree.

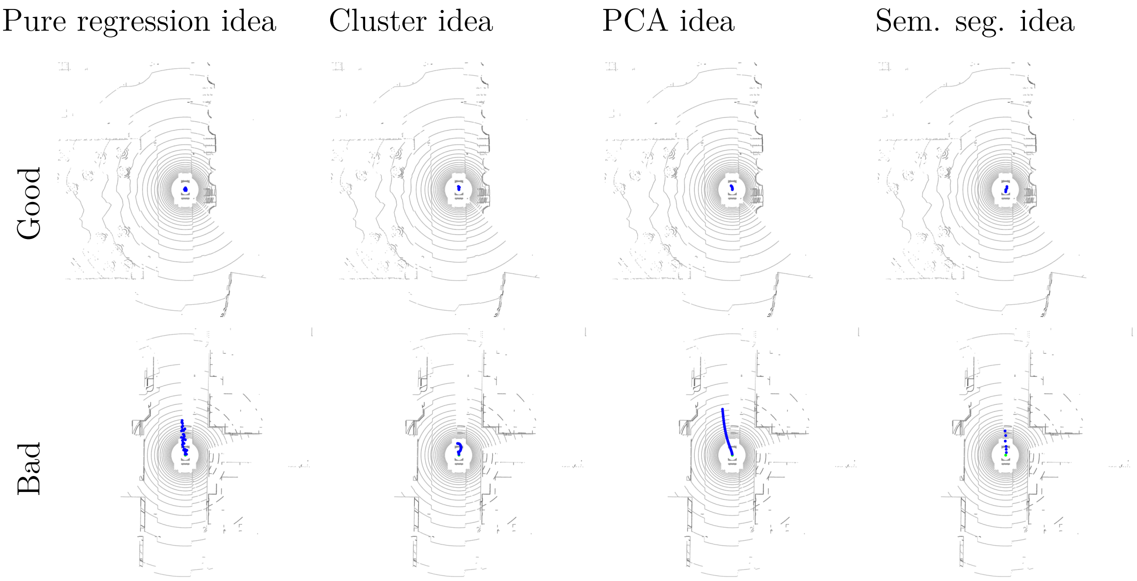


Figure 5.12: Waiting for traffic light. Top and bottom: Vehicle is waiting for green light and networks should predict no movement. Evidently, the networks manage to make appropriate predictions in some cases but not in others.

Another interesting result can be seen in Figure 5.13 where the models predict undesired turns. In the top row, all networks want to take a left turn into a building. The intention direction at this time step was left, but the distance to the turn was more than 100 m.

In the bottom row it can be seen how the networks predict an early right turn, where the intention indeed is a right turn. Just before the turn, there is a red traffic light which the car slows down for, presumably causing the network to predict the early turn.

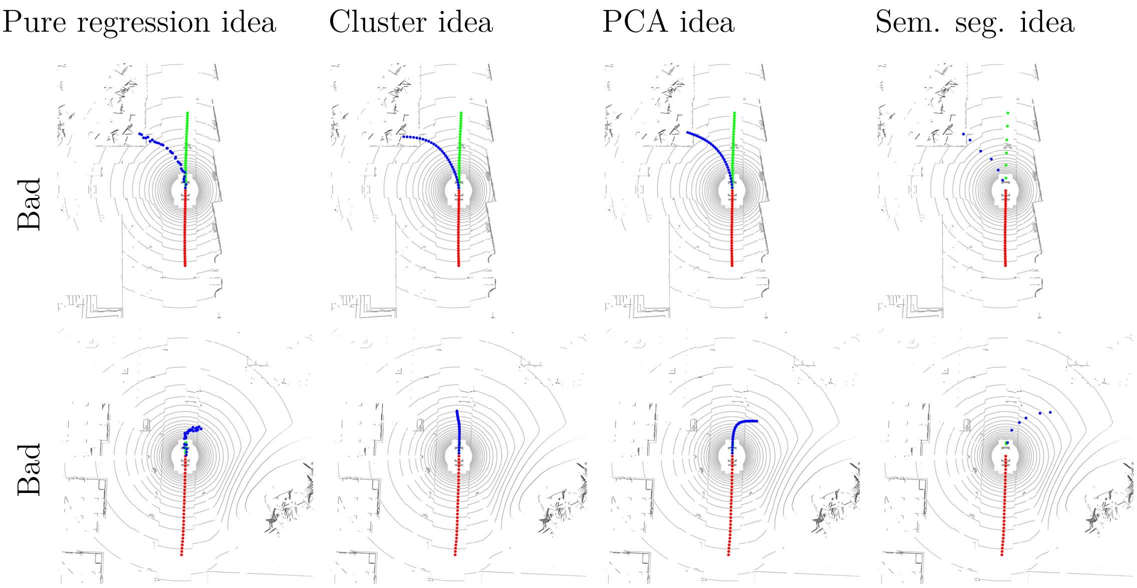


Figure 5.13: Another vehicle is driving in front of the car, and thus the ego-vehicle in the recorded data is slowing down. This scenario causes the predictions to behave undesirable. Top: wanting to turn when it in fact should continue straight ahead. Bottom: wanting to turn when it in fact should stop for a traffic light. A reasonable explanation for this behaviour is that slowing down near a crossroads usually means that the vehicle will soon be turning. In the scenario shown, the turn is expected, but more than ten meters ahead of the predicted trajectory.

As could be seen in Figure 5.12, the networks incorrectly predicted movement while waiting for some of the traffic lights. More interestingly, in some situations, the networks showed to behave as desired when a traffic light changed from red to green, see Figure 5.14. No movement is predicted while waiting. As soon as the traffic light turns green, i.e., at the first time step with green light, the network predicts a small movement and continues to increase the length of the trajectory for each time step. At the time the car starts moving, the predicted trajectory aligns remarkably well with the ground truth. Figure 5.14 displays this behaviour only for the pure regression network *More layers 1*, but similar behaviour was found for the other networks as well, although minimal for the semantic segmentation network.

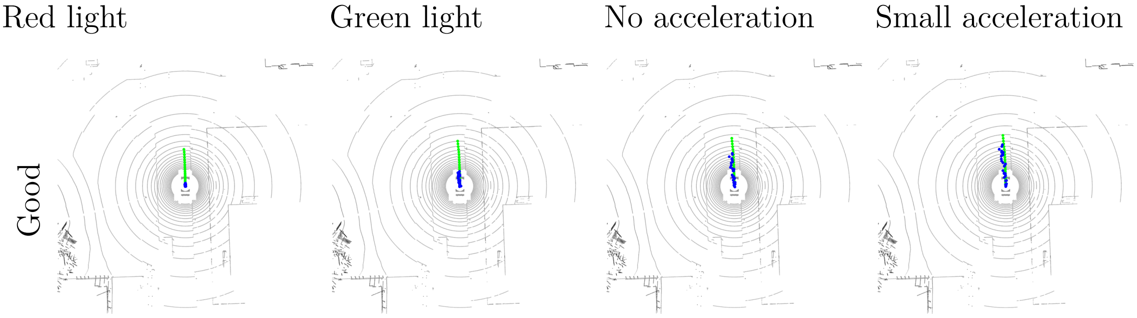


Figure 5.14: Series of images depicting how the pure regression network *More layers 1* predicts movement as soon as the traffic light turns green. First image: Car is waiting for a traffic light to turn green, predicting no movement. Second image: Traffic light changed from red to green, predicting a small movement even though the acceleration is zero. Third image: Predicted trajectory grows longer, still with no acceleration present. Fourth image: Finally, the car starts moving and the predicted trajectory aligns with the ground truth.

In contrast to the ability discussed above, where the network clearly predicts acceleration before initial movements of the car, the following example demonstrates a more inconclusive behaviour. In the situation of a right turn without intention, the network seems to predict turning too late, see Figure 5.15. The acceleration and steering angle have been plotted for the same time period in Figure 5.16. It can be seen in Figure 5.15 that the predicted turn is initiated at time step 2147 but then aborted at time step 2149. Between time steps 2149 and 2157, the trajectory becomes better and better. However, the turn is initiated when there is little change in acceleration and steering angle (during time steps 2150-2155). It then rapidly changes (during time steps 2156-2157) into the more accurate prediction at time step 2157, for which the acceleration and steering rate are considerably higher. Thus, the network seems to take initiative without prior turning movement, but that it benefits heavily from it.

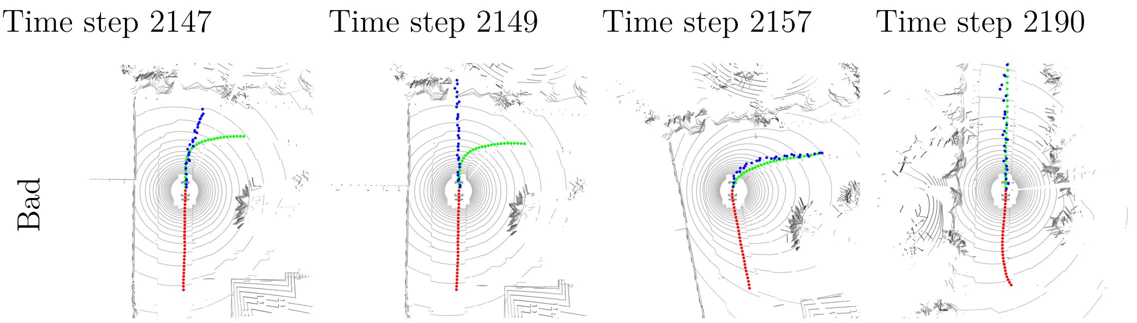


Figure 5.15: Series of predictions in a turn without intention, using the pure regression network *More layers 1*. The first (left) image shows how the turn is being initiated in the prediction of time step 2147, although too slowly compared to the ground truth. Just two time steps later, at time step 2149, the turn has been aborted and the predicted trajectory continues straight off the road. Between steps 2150 and 2157, the predictions become better and better and finally align quite well with the ground truth. The network continues to predict with about the same accuracy, however with some overshooting, until the curve has ended.

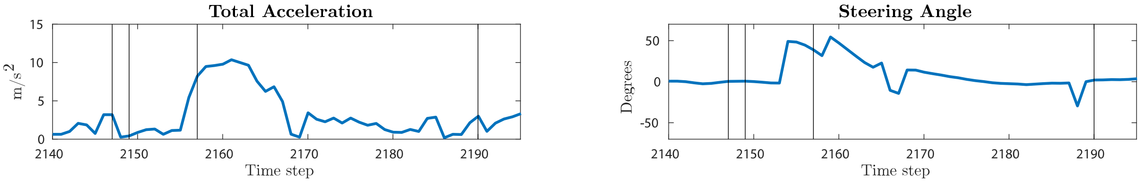


Figure 5.16: The total acceleration and the steering angle have been plotted over the time series depicted in Figure 5.15. The four vertical lines in each plot mark the time steps corresponding to the images in Figure 5.15. It seems like the network predicts turning trajectories based on these signals rather than the lidar image, although it is unclear to what degree.

The car should stop when approaching another car which is standing still. As seen in the bottom row of Figure 5.17, the models do not capture this, and instead predicts driving through a car which has stopped. This can be seen by comparing the ground truth trajectory with the predicted one, as the predicted is about twice as long. On the other hand, when the other car is having some speed, the future trajectory could possibly go through it. The models seem to behave accordingly, although it should also be mentioned that it is impossible to know the speed of other objects without at least one top view lidar image from another time step. For this reason, it is not surprising that the networks do not take the speed of other cars into account.

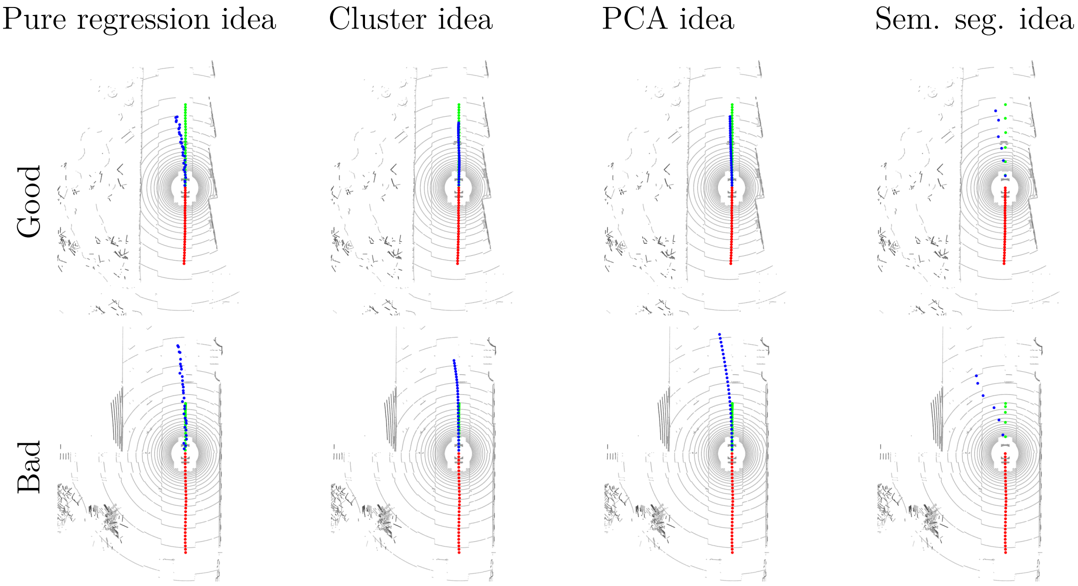


Figure 5.17: Networks predict trajectories intersecting other vehicle. Top: The other car is moving and the predicted trajectory is correctly going through it. Bottom: Other car is standing still but the network incorrectly wants the car to go through it. Note that more lidar images are required for the network to figure out if the car in front is moving or not.

The network *More layers 2* which was trained for a total of 63 hours did reach a lower test loss, but seems to make the same kind of errors as when trained for 14 hours only. Some examples can be seen in Figure 5.18, where the major improvement seems to be the smoothness of trajectories. In column three of the same figure, the network seems to be better at correcting for the perturbed steering signal.

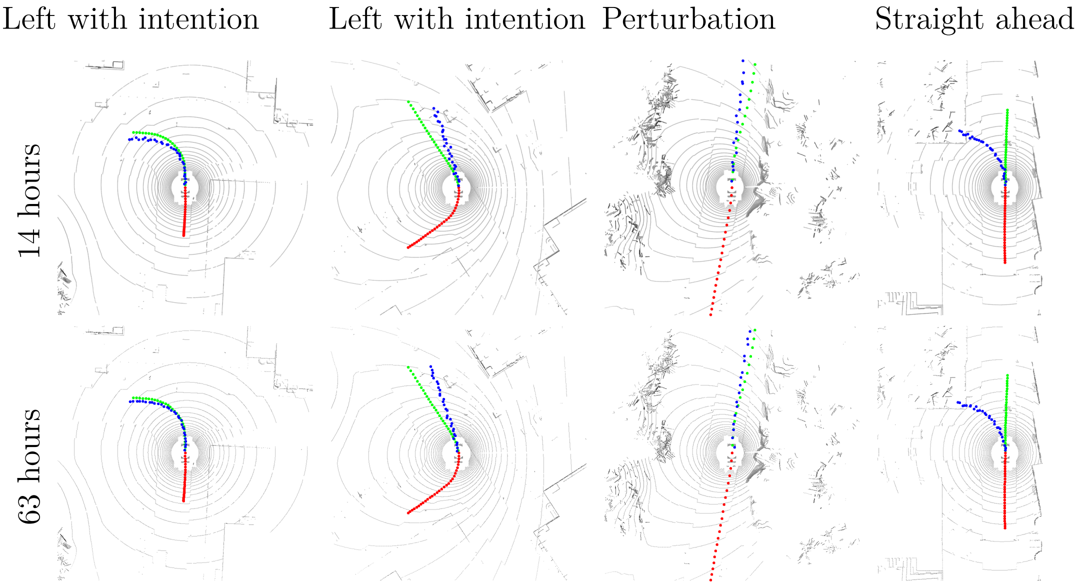


Figure 5.18: Comparison between predictions by *More layers 2* with different training times. Top: Trained for 14 hours. Bottom: Trained for 63 hours. The networks seems to be making the same mistakes, although predicting smoother trajectories in general.

A final note on the semantic segmentation network is that its MSE is much larger compared to the other networks, but that it still seems to learn something meaningful. Figure 5.19 shows how the predictions farther into the future are moving forward with respect to the car and that turning left seems to be more likely, as the high probability area tends to extend to the left.

5.3 Benchmarking

One of the benefits of using CARLA is the ability to benchmark driving models using the same metrics. Although the MPC by itself worked as expected, controlling the car in CARLA did not work. When testing the MPC on the ground truth trajectories (which by definition are correct), it very rarely predicted positive throttle values and more often positive brake values. This resulted in the car not driving anywhere. CARLA is implemented in such a way that during the first two to three seconds of driving, the car is unable to apply control signals. To overcome any possible issues related to this, it was also tested to drive using some initial positive throttle values, as to not lag in the beginning. This did not help in any way, and the car stopped as soon as the throttle signal was controlled by the MPC again. As the MPC did not function in CARLA on the ground truth trajectories, the performance of the models could not be measured in the simulator. See Section 6.4 for a discussion on possible causes.

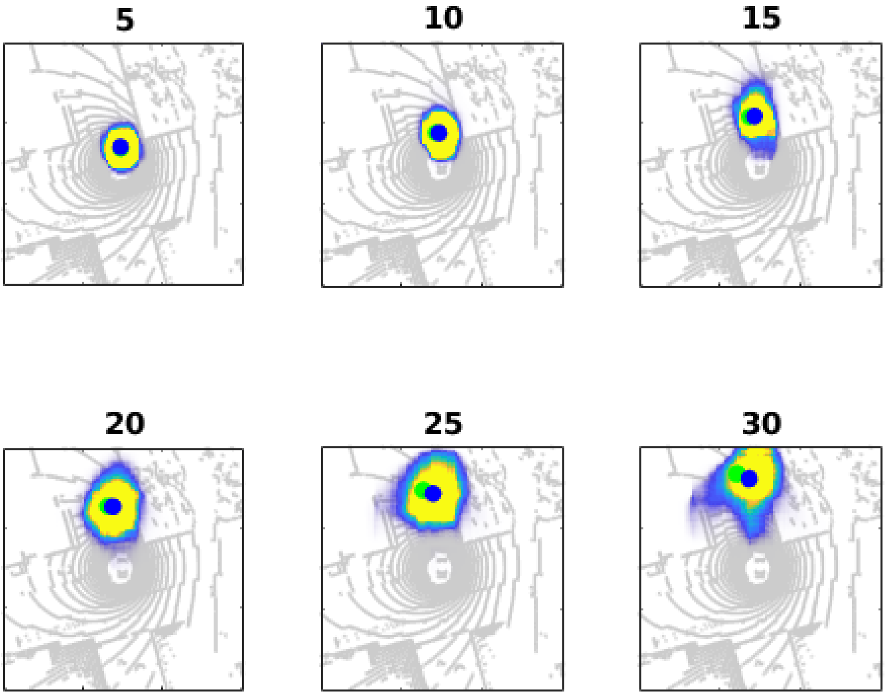


Figure 5.19: Predicted segmentation map from the semantic segmentation network at different time steps (5, 10, 15, 20, 25 and 30) into the future. Bright yellow means high probability and purple means low probability. The probability values control the transparency of the segmentation map, making the top view lidar image visible in the background. As for the other prediction plots, the green dot is the ground truth position and the blue dot is the predicted position, calculated as the center of mass of the semantic segmentation map.

6
Discussion

This chapter will analyze and debate the results, which unfortunately performed poorer than hoped. The architecture of the networks and the ideas in general will be discussed in light of the results. Moreover, other factors which might have contributed to the result will also be examined and discussed.

In Section 6.1, 6.2, and 6.3 are the structure of the neural networks and the results presented. It is followed by a discussion about the difficulties integrating the MPC in the CARLA benchmark in Section 6.4, and finally the direction for future work is treated in Section 6.5.

6.1 Predictions

The models presented in Section 5 (Results) did not perform as good as expected. Because of the promising results in [6], which could predict future vehicle positions, the leap to predicting future positions at specific time steps did not seem very large. In contrast, the results show that this problem is much more difficult and requires further investigation.

One of the most prominent problems is that the predictions often are too far from the ground truth. It is not necessary for the predictions to perfectly align with the ground truth, as human drivers do not drive along the exact same path every time they take a turn. However, the visual inspection of predicted trajectories indicate that none of the networks reach acceptable performance. It is surprising that even the most simple situations, such as following the road as it turns, is a big issue for all networks. The main cause for this seems to be the fact that the networks do not rely on the lidar images as much as the IMU-data. It is possible for a human to look at the lidar images and tell that the road is turning, meaning that the necessary information is there.

In Figure 5.14 one can see that the network seems to ”take initiative” and starts to predict movement as the traffic light status changes. In contrast to simply predicting to follow the current trajectory with the current speed and acceleration, which is an easy problem, this shows a deeper understanding of the environment. However this ”initiative” is somewhat lacking in turns without intention, i.e., where the road is curved. As seen in Figure 5.15, until the turn is noticeable in the input steering angle, the predictions are unstable and might shift greatly and suddenly. Once the car has started to turn, the predictions quickly change towards the correct trajectory. But as mentioned before, following a trajectory is a simple task compared to taking the initiative to turn.

It also seems like the models do not infer enough information from the lidar images, but instead depend excessively on the IMU data. They mostly prefer to follow the trajectory inferred from the IMU data. This is visible in Figure 5.13 where the ground truth is driving slower than usual due to a vehicle in front. The model seem to infer a turn from the lowered velocity, even though the intention and lidar image says otherwise. Perhaps would more examples of these kinds of scenarios in the data counteract this wrongful inference.

6.2 Data

When working with machine learning tasks in general, it is reasonable to ask the question whether the amount of training data is sufficient and if it covers all the cases expected to be handled by the model? As the training data was balanced so that the major situations (turning left, approaching traffic light, etc.) where represented in equal amounts, some effort was put into making the ratios more even. However, the models would most certainly benefit from having larger amounts of the situations occurring more seldom in the simulator. It is possible to start driving and recording close to left turns and right turns specifically, although the recorded data would have to be trimmed as the exact starting location cannot be set in CARLA. Furthermore, it is not possible to control the traffic lights, pedestrians or other vehicles, making the data collection of specific situations hard.

Another question relevant for this project is whether lidar and IMU data contains enough information to properly control the vehicle. Unlike most autonomous driving systems, a camera was not used which is an unusual decision. However, [6] used lidar as a main source of information (without any additional cameras) and obtained good results. Furthermore, it is possible for a human to identify the road and most other objects using the lidar top view images. Thus, it is reasonable to believe the input used contained enough information to derive good driving decisions. Nonetheless, it is probable that an additional, traditional, camera would improve the results, as more data and other types of features usually yield better performance in neural networks.

The data was also very tainted by noise. As the data was recorded, uniform noise was added to the steering signal of the autopilot, which affected all collected data. This was done with the motivation to not only make the model noise resistant and robust, but also make the model learn how to correct for bad positioning on the road. By intentionally feeding the models with slightly zigzag driving data, the model would hopefully be a more sturdy driver and be able to correct the trajectory if it was a little bit off. A natural consequence of adding noise would be an increased difficulty in filtering out the essential information, increasing the needed training time.

Another concern has been the size of the cities in CARLA. While Town01 (used to create the training set) is clearly the larger one, both are very small. As such, the data was highly limited to the few scenarios offered in the towns. Even if the actions and positions of the dynamic objects, i.e., other vehicles and pedestrians, varied, the scenes remained the same, making the data ”easier” than real data. A natural implication of this would be overfitting of the models — but this is not observed. The somewhat unexpected lack of overfitting is believed to occur due to the architecture of the neural networks.

6.3 Architectures and Ideas

Judging from the validation plots, none of the networks seems to overfit. This is also true for the training loss, indicating that further training will improve performance, although at a slower and slower pace. Why the models do not overfit even though the small towns facilitate overfitting, is believed to be because of spatial dropout. It is used throughout all the networks, which leading purpose is to counteract overfitting.

Nevertheless, it is also possible the network architectures simply lack enough complexity to fully capture the essence of predicting correct trajectories. This suggestion is supported by the fact that the networks *More Layers 1* and *More Layers 2*, which have deeper architectures, produce a lower loss. On the other hand, the networks consist of parameters in the order of 10 millions, which seem to be enough for [4] and [6], although these solve slightly different tasks.

Another common issue in machine learning projects is the amount of training time. How much is enough to get the desired results and when is it reasonable to stop? Except for the experimentation phase, where the training time differed vastly between architectures, the set training time was 7 or 14 hours. In the context of autonomous driving, this is very meager. It was observed that training *More layers 2* (which had the smallest test loss) for a total of 63 hours did not make any significant difference in the problematic situations. This indicates that the major problem does not lie in the amount of training time, but that there is some other information required to produce better predictions.

For all networks except the semantic segmentation idea networks, $n_p = n_f = 30$. It is possible that predicting 30 time steps (i.e., three seconds) into the future is to ask to much of the networks. On the other hand, it seems like the networks do not struggle with the number of time steps, but rather the overall type of trajectory. For example, as seen in Figure 5.13, the trajectories look rather realistic, but are not close to the ground truth, indicating that the number of predicted time steps is within reasonable limits.

Moving on to discuss the specific architectures, the RNNs did seem to learn, but not to the point where the predictions could be considered useful. This is clear since the validation loss is well over the baseline. It is possible that further training would result in useful models, although the slow training speed did not motivate continuing. It is not clear why the recurrent networks perform worse than the rest, although it could be due to the different way of training. It could also be that there is something wrong with the implementation of the networks, as much time was spent on making them trainable at all.

The implementation of the semantic segmentation idea does have a disadvantage against the other ideas, which is that predictions can only be made within the region of interest. This means that locations along the ground truth trajectories which occur more than 30 m ahead of the vehicle will not be present in the ground truth semantic map, as the width and height dimensions of the semantic maps are fixed. This is a possible cause for the high test loss observed. The problem can be alleviated using ground truth semantic maps that stretch outside the region of interest. However, the lidar images are cropped to the region of interest, forcing the network to then make uninformed guesses of future locations.

The inferior performance of the cluster idea was surprising, as the idea behind it seemed very promising (explained in Section 3.3.3). One reason it learns slower than the other neural networks seems to be its more complex backpropagation. It learns two tasks simultaneously, one classification task and one regression task. The backpropagation is also class specific, i.e., trains on only the correct class. In contrast to class agnostic backpropagation, i.e., training on all classes simultaneously, a class specific approach is likely less efficient.

Next, the performance of the PCA idea and the pure regression idea will be discussed. Both are regression problems but with different number of outputs. As fewer outputs were thought to be easier, their performance do not match the expectations. It might be that the PCA idea with fewer variables are more sensitive to small differences. Getting only one variable wrong offsets all points instead of just one. However, despite the slightly better loss in the pure regression idea, the performance of the PCA idea is superior. It produces nice and smooth curves which are much more useful than the cluttered predictions made using pure regression.

In conclusion, no network architecture seems to outperform the others significantly and no network is close to overfitting. Therefore there is no reason to believe another type of architecture or idea would solve the current problem. The problem is most likely in one of the other mentioned factors, primarily the amount of training time, the amount of data, and the amount of desired scenarios present in the data.

6.4 Benchmarking Problems

The models could not be benchmarked in CARLA, due to problems using the MPC. Note that the MPC could in fact follow a given trajectory with great precision, using its internal bicycle model. One of the possible causes for not working in the simulator could be that the neural network constructing throttle and brake signals, given a desired acceleration, may not work as desired. It can be seen in figure 4.13 that, although prediction accuracy varies between different time steps, the network seems to respond with throttle when acceleration is desired and brake when deceleration is desired. Hence, it is not very likely that this is the main cause of failure.

A more likely cause of failure may be that there was something wrong with the integration of the MPC into CARLA. However, no such faults could be discovered during the project. It is unfortunate that the benchmarking could not take place, as much time and effort was spent by Juliano and Dapeng on constructing the MPC. Also, a lot of time was spent on discussing the possibilities and options for the MPC, constructing the object distance maps, and so on.

6.5 Future Work

It is evident that the results can be improved. One possible key to better performance is using lidar readings from multiple time steps as input. This was tested during the project, but as the training time was increased substantially, while maintaining similar performance, it was not investigated further. However, training for a longer period of time might increase performance and should help the network get a better sense of dynamic objects in the environment. Moreover, adding camera images to the input might help alleviating the same problem, although the training speed would decrease even more. The networks might then learn when to slow down, stop and start using visual input instead of relying on the acceleration measurements.

A large portion of the data was recorded while driving along straight roads and waiting for traffic lights to turn green. Although the data was balanced according to the different situations, the training data could be improved by recording more turns specifically. Also, the critical scenario of stopping for other vehicles in front of the ego-vehicle could be extracted as another category and balanced accordingly. It would also have been reassuring to get the network to overfit to the training data. This would confirm that the network learns correctly and that there is nothing inherently wrong with the network. By training on a very small fraction of the training data, one should be able to force the network to overfit.

The system could not be tested in CARLA, which would be one of the most interesting things to accomplish in the future. This would enable the direct perception approach used in the project to be compared with other approaches and systems, using the same benchmarking suite.

As only the best pure regression idea network was trained for 63 hours, it would also be interesting to see how the other network ideas would perform when given more training time.

Finally, we hope our work will inspire others to explore different techniques in combination with direct perception to control self-driving cars.

7 Conclusion

The purpose of the thesis was to evaluate the performance of different deep neural network architectures as they control a self-driving car. Their inputs are top view lidar images of the surroundings, IMU-values and other sensors values of a simulated car. Their output is a trajectory which can be used by other control mechanisms, while still allowing for human interpretation. This is referred to as direct perception.

Four different main ideas were tested, incorporating regression, clustering, principal component analysis and semantic segmentation. Multiple networks belonging to each idea were implemented with variations. One of the major findings was that the best network architecture of each idea perform similarly good, with the exception of the semantic segmentation idea, which did not perform as good as the rest.

The results indicate the networks learned, to some degree, to predict feasible trajectories. However, the trajectories are not satisfactory and are far from usable in real vehicles. On the other hand, the trajectories are easily interpretable by humans and could thus be used to reason about what the networks struggle to learn. The problematic scenarios include predicting trajectories for taking turns where no intention is given, i.e., the road turns and the car must simply follow it, driving straight ahead when turning directions are present but intended for crossroads farther away, stopping for traffic lights, and avoiding other vehicles. Scenarios which the networks do seem to handle better, yet not completely satisfactory, are predicting trajectories near crossroads where the turning direction is given, beginning to move after waiting at traffic lights and driving on a straight road. The networks implementing the PCA idea have somewhat higher test loss compared to the overall best network, which is a pure regression network. On the other hand, the trajectories of the PCA idea networks are more realistic and are thus preferable. The performance of the networks is likely to increase from further training, although the results indicate that insufficient predictions are prominent even after 60 hours of training. More training data, specifically of the problematic situations mentioned, could help increase the performance further. Changing the network architectures does not seem to yield vastly different results, but could provide some fine tuning, should these major issues be solved. It cannot be concluded from the results whether lidar and GNSS-IMU sensors provide enough information to successfully perform the direct perception task.

In consequence of the findings, obtaining satisfactory performance of a neural network for direct perception systems seems harder than expected. However, we conclude that the techniques in this project have great potential and that future research might yield positive results.

Bibliography

[1] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, “Deepdriving: Learning affordance for direct perception in autonomous driving,” in *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec 2015, pp. 2722–2730.

[2] J. Janai, F. Güney, A. Behl, and A. Geiger, “Computer vision for autonomous vehicles: Problems, datasets and state-of-the-art,” *CoRR*, vol. abs/1704.05519, 2017. [Online]. Available: http://arxiv.org/abs/1704.05519

[3] H. Zhu, K. Yuen, L. Mihaylova, and H. Leung, “Overview of environment perception for intelligent vehicles,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 18, no. 10, pp. 2584–2601, Oct 2017.

[4] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, “End to end learning for self-driving cars,” *CoRR*, vol. abs/1604.07316, 2016. [Online]. Available: http://arxiv.org/abs/1604.07316

[5] P. Drews, G. Williams, B. Goldfain, E. A. Theodorou, and J. M. Rehg, “Aggressive deep driving: Combining convolutional neural networks and model predictive control,” in *Conference on Robot Learning*, 2017, pp. 133–142.

[6] L. Caltagirone, M. Bellone, L. Svensson, and M. Wahde, “LIDAR-based driving path generation using fully convolutional neural networks,” in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, Oct 2017, pp. 1–6.

[7] G. Bresson, Z. Alsayed, L. Yu, and S. Glaser, “Simultaneous localization and mapping: A survey of current trends in autonomous driving,” *IEEE Transactions on Intelligent Vehicles*, vol. 2, no. 3, pp. 194–220, Sept 2017.

[8] J. Aulinas, Y. Petillot, J. Salvi, and X. Lladó, “The SLAM problem: A survey,” in *Proceedings of the 2008 Conference on Artificial Intelligence Research and Development: Proceedings of the 11th International Conference of the Catalan Association for Artificial Intelligence.* Amsterdam, The Netherlands: IOS Press, 2008, pp. 363–371. [Online]. Available: http://dl.acm.org/citation.cfm?id=1566899.1566949

[9] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “CARLA: An open urban driving simulator,” in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[10] K. Schmid, K. Waters, L. Dingerson, B. Hadley, R. Mataosky, J. Carter, and J. Dare, “Lidar 101: An introduction to lidar technology, data, and applications,” *NOAA Coastal Services Center*, 2012, revised.

[11] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 1998.

[12] S. Haykin, *Neural Networks and Learning Machines*, ser. Pearson International Edition. Pearson, 2009. [Online]. Available: https://books.google.se/books?id=KCwWOAAACAAJ

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, http://www.deeplearningbook.org.

[14] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, “Fast and accurate deep network learning by exponential linear units (ELUs),” in *International Conference on Learning Representations (ICLR)*, 2016.

[15] S. Ruder, “An overview of gradient descent optimization algorithms,” *CoRR*, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[16] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *International Conference on Learning Representations (ICLR)*, 2015.

[17] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, “Object recognition with gradient-based learning,” in *Shape, Contour and Grouping in Computer Vision.* London, UK, UK: Springer-Verlag, 1999, pp. 319–. [Online]. Available: http://dl.acm.org/citation.cfm?id=646469.691875

[18] F. Yu and V. Koltun, “Multi-scale context aggregation by dilated convolutions,” in *International Conference on Learning Representations (ICLR)*, 2016.

[19] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *CoRR*, vol. abs/1412.3555, 2014. [Online]. Available: http://arxiv.org/abs/1412.3555

[20] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[21] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, Nov 1997.

[22] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” in *International Conference on Learning Representations (ICLR)*, 2015.

[23] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[24] J. Tompson, R. Goroshin, A. Jain, Y. LeCun, and C. Bregler, "Efficient object localization using convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 648–656.

[25] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 33–55, March 2016.

[26] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli, "Kinematic and dynamic vehicle models for autonomous driving control design." in *Intelligent Vehicles Symposium*, 2015, pp. 1094–1099.

[27] S. Rogers and M. Girolami, *A First Course in Machine Learning*, 2nd ed. Chapman & Hall/CRC, 2016.

[28] E. Shelhamer, J. Long, and T. Darrell, "Fully convolutional networks for semantic segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 4, pp. 640–651, April 2017.

[29] H. Noh, S. Hong, and B. Han, "Learning deconvolution network for semantic segmentation," in *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec 2015, pp. 1520–1528.

[30] L. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 834–848, April 2018.

[31] D. Barnes, W. Maddern, and I. Posner, "Find your own way: Weakly-supervised segmentation of path proposals for urban autonomy," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 203–210.

[32] F.-F. Li, A. Karpathy, and J. Johnson, "Lecture notes 8 in convolutional neural networks for visual recognition," February 2016.

[33] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, June 2014, pp. 580–587.

[34] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[35] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[36] S. Gugger, "The 1cycle policy," accessed 2018, July 9. [Online]. Available: https://sgugger.github.io/the-1cycle-policy.html#the-1cycle-policy

[37] ——, “How do you find a good learning rate,” accessed 2018, Apr. 18. [Online]. Available: https://sgugger.github.io/how-do-you-find-a-good-learning-rate.html

A

Lidar implementation in CARLA

At the beginning of this project CARLA did not support any lidar functionality. As the main data intended to be lidar, a system was implemented to extract lidar data from the existing sensors in CARLA. About half-way through the project CARLA released their own lidar solution, a ray-cast based lidar sensor available natively in the simulator. As their implementation was both faster and more true to the nature of real lidar systems than ours, it was used throughout the remainder of the project. However, since a lot of time and energy was put into this functionality, the process will be described here.

A.1 CARLA's depth maps

Before their lidar sensors, CARLA used *depth cameras* to record the distance to objects. The images called *depth maps* used the RGB values to encode a distance value to each pixel. In Figure A.1 a depth map from CARLA can be seen. Even though the colors are distorted one can easily perceive the settings. Four of these depth cameras were placed above the car facing forwards, right, left and backwards, which together covered 360 degrees. The value in a pixel of the image captured by a depth camera corresponds to the distance between the camera and the nearest object point seen through that pixel in the near near clip plane. The near clip plane is the gray plane which can be seen in Figure A.3.

A.1.1 Sampling the relevant points

With the recorded depth maps it is not possible to convert the 2D data into a 3D point cloud. In order to convert the four depth maps into a lidar point cloud, the relevant pixels need to be known. The relevant pixels is which pixels on the depth map the lidar rays would have hit. Furthermore, the angle of the ray to each respective pixel must also be known to create a 3D point cloud.

The relevant pixels and their respective angle is calculated by simulating a lidar, which in retrospect sweeps over the recorded depth images. An imagined lidar and it's rays is put in front of a depth map. The lidar rays is set to mimic a Velodyne HDL-32. The Lidar starts at the left side and sweeps horizontally over the image by rotating around the y-axis. For every 0.016°the intersecting pixels (and corresponding angles) are saved. In Figure A.2 one can see the positions of the relevant pixels together with the desired curvature.

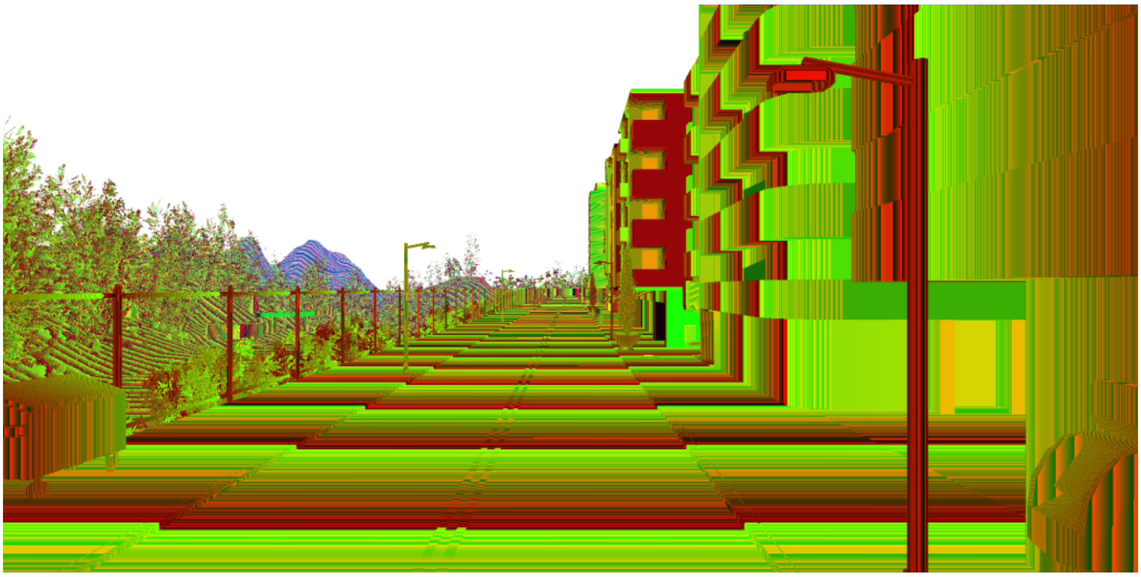


Figure A.1: Depth map recorded by the CARLA depth cameras. The RGB values represent the distance to each pixel.

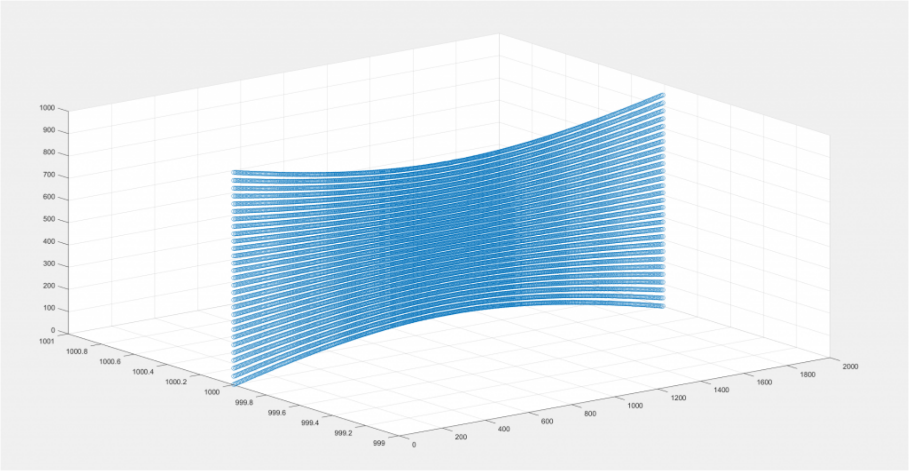


Figure A.2: The blue dots is the relevant pixels which should be extracted from each depth map in order to recreate a lidar point cloud. The curvature arises when a ray sweep a perpendicular plane.

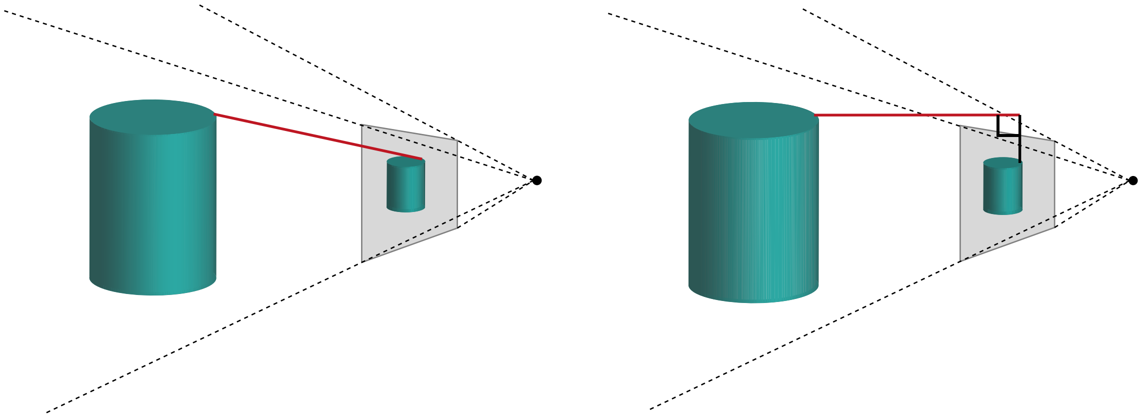


Figure A.3: The image shows which point of an object each pixel "sees". Traditionally the distance to an object is measured between the pixel and the object, red line in the left image. Instead the depth value in CARLA is decided by the perpendicular distance between the object and the pixel, the red line in the right image.

A.2 converting to a 3D point cloud

For every depth map, the depth values are recorded in the RGB image at the positions of the relevant pixels. The depth values together with the angles can then be used to define 3D points in a polar coordinate system. These are then converted to cartesian coordinates and a final correction is applied, namely

$$[x, y, z] = \frac{[x, y, x]}{cos(h)cos(v)}, \tag{A.1}$$

where h is the horizontal angle and v is the vertical angle in the polar coordinates. The resulting point cloud can be seen in A.4. The last correction is needed because CARLA's depth cameras measure depth in an unorthodox fashion. In contrast to how depth cameras work in general, each pixel of the CARLA depth map contains the perpendicular distance between the object point, seen through that pixel, and the xz-plane in which the camera is positioned. An illustration of this can be seen in Figure A.3.

A.3 Interpolation

An artifact of the depth camera to lidar conversion was that the resolution in the height and width dimensions affected the outcome of the point cloud. As the resolution shrunk smaller, the quantization error grew larger. This problem protruded especially for low resolutions in the height of the image, i.e. one pixel in height represented several meters in depth far away. To remedy this, values in the points of interest were linearly interpolated, producing smoother results. However, as a side effect, neighbouring pixel values with great difference resulted scattered the point

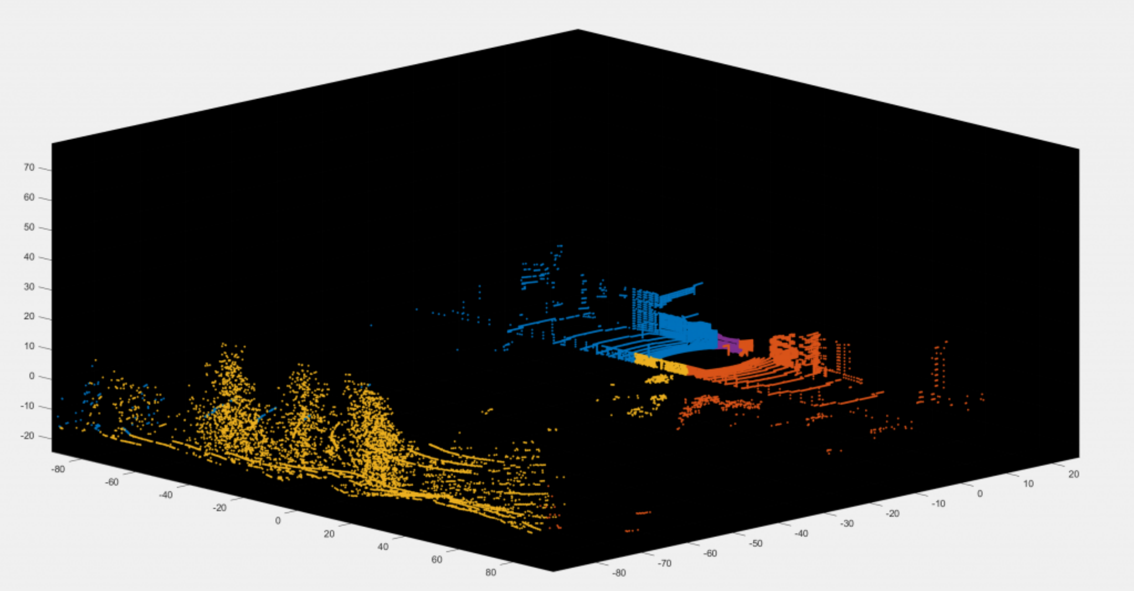


Figure A.4: The final result of the lidar extraction form the depth cameras. The colors represent which camera the points are recorded with.

cloud. In turn, interpolation was only applied where the value difference between neighbouring pixels was smaller than some threshold t, resulting in more realistic points clouds.

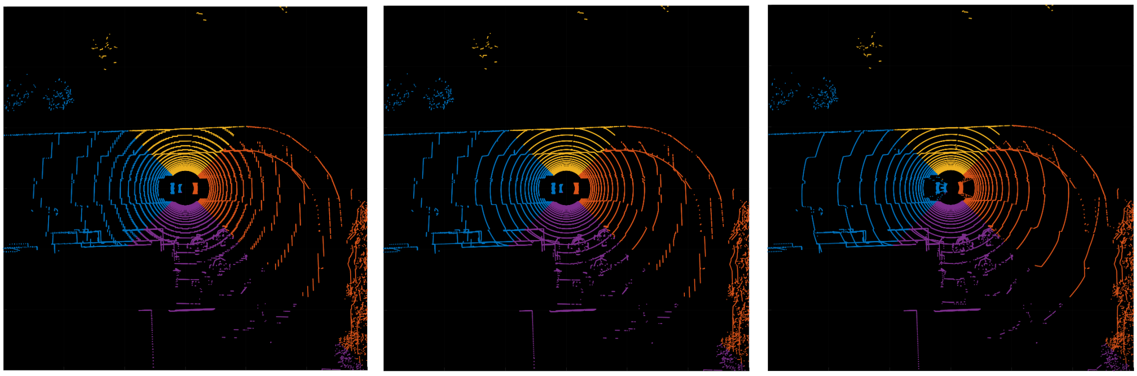


Figure A.5: A depth camera with resolution 400x300 was used. The difference in smoothness of the lines between not using interpolation (left) and using interpolation with a threshold of 1 meter (middle) and using a threshold of 2 meters (right) is clearly visible.

B

Pure Pursuit

Pure pursuit is a control technique, like MPC, which was first considered due to its simplicity. However, it is used for path stabilization rather than trajectory stabilization, thus it does not control the velocity of the vehicle. This restriction does not match the aim of the project and was only used initially to get a fully working pipeline from input to steering signals.

[25] evaluate the performance of different path stabilization techniques and show that other path stabilization techniques are superior to pure pursuit in terms of performance, but pure pursuit is considerably simpler than the other techniques. In order to calculate the steering signals along a given path, pure pursuit makes use of the vehicle's (rear wheel) position (x, y) in the global coordinate system, the vehicle's heading angle (yaw) θ and a predefined look ahead distance L from the vehicle position to find a reference position along the path. By fitting a semicircle (see Figure B.1) through the vehicle position and the reference position, the desired steering rate ω is calculated for the current velocity v of the vehicle [25], using the formula

$$\omega = \frac{2v\sin(\alpha)}{L}, \tag{B.1}$$

where the angle α is found through

$$\alpha = \arctan\left(\frac{y_{ref} - y}{x_{ref} - x}\right) - \theta. \tag{B.2}$$

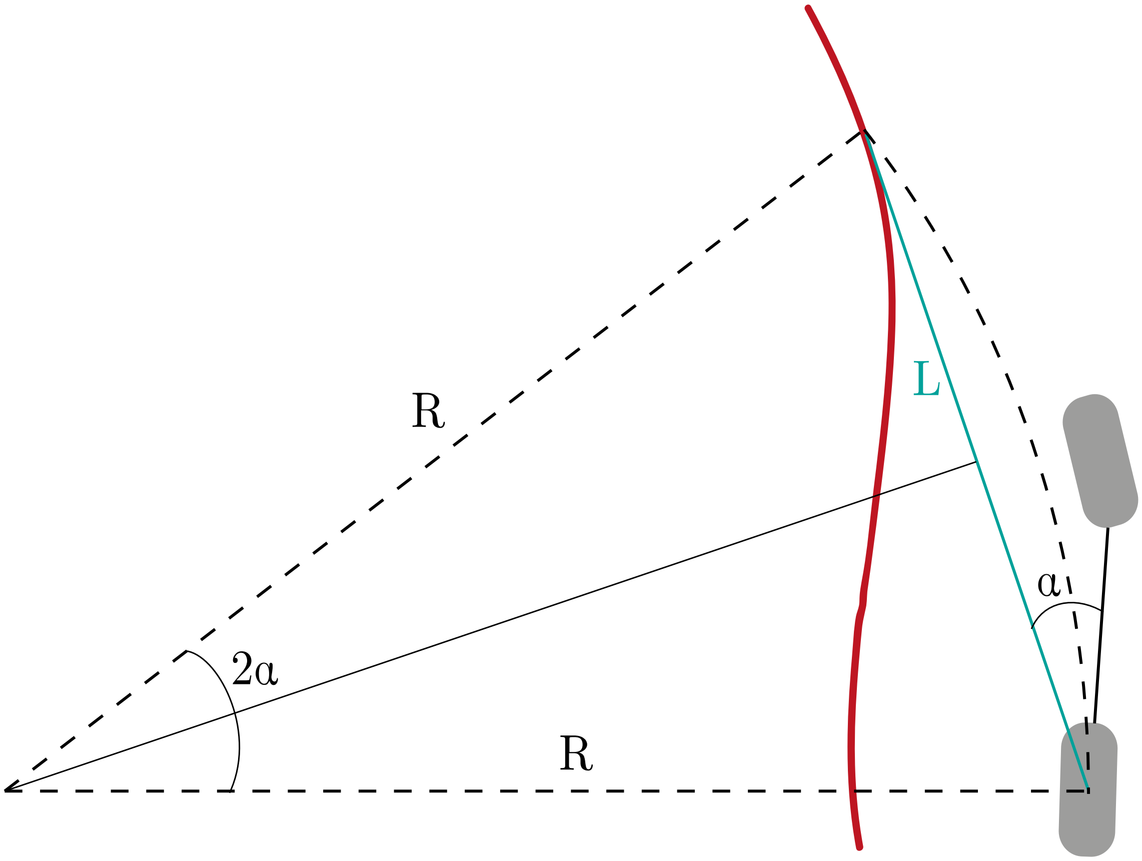


Figure B.1: A semi circle is fitted between the current position of the rear wheel and the look ahead distance L. The steering rate ω is decided by eq. B.1.

C

Recorded measurements from CARLA - an Example

platform__timestamp	game__timestamp	location__x	location__y	location__z	acceleration__x	acceleration__y	acceleration__z
1269707750.00000000	100.00000000	338.97998047	301.25997925	40.09062576	0.00000000	0.00000000	0.00000000
1269708125.00000000	200.00000000	338.97998047	301.25997925	40.03348541	-0.00000000	-0.00000001	0.00000000
1269708500.00000000	300.00000000	338.97998047	301.25997925	39.87845993	-0.00000000	-0.00000001	0.00000000
1269708750.00000000	400.00000000	338.97998047	301.25997925	39.62564468	-0.00000000	-0.00000001	0.00000000
1269709125.00000000	500.00000000	338.97998047	301.25997925	39.27513885	-0.00000000	-0.00000001	0.00000000
1269709375.00000000	600.00000000	338.97998047	301.25997925	38.83083344	0.00000000	0.00025759	-0.00000000
1269709750.00000000	700.00000000	338.97949219	301.26028442	38.67161179	0.00000072	0.17343378	-0.00015492
1269710000.00000000	800.00000000	338.97836304	301.26092529	38.69230652	-0.00000068	-0.16834582	0.00013994
1269710375.00000000	900.00000000	338.97830200	301.26080322	38.71649170	-0.00000022	-0.03298739	0.00008213
1269710625.00000000	1000.00000000	338.97863770	301.26055908	38.73567581	0.00000009	0.00225859	-0.00002303
1269710875.00000000	1100.00000000	338.97885132	301.26031494	38.74888611	0.00000006	0.00745106	-0.00002318
1269711250.00000000	1200.00000000	338.97909546	301.26022339	38.75733185	0.00000002	0.00567119	-0.00001154
1269711500.00000000	1300.00000000	338.97927856	301.26010132	38.76248550	0.00000001	0.00455019	-0.00000555
1269711875.00000000	1400.00000000	338.97949219	301.26004028	38.76554108	-0.00000000	0.00289266	-0.00000228
1269712125.00000000	1500.00000000	338.97949219	301.26004028	38.76730347	0.00000000	0.00159205	-0.00000088
1269712500.00000000	1600.00000000	338.97943115	301.26004028	38.76830292	0.00000000	0.00117235	-0.00000040
1269712750.00000000	1700.00000000	338.97949219	301.26004028	38.76857376	0.00000000	0.00107997	-0.00000018
1269713125.00000000	1800.00000000	338.97943115	301.26004028	38.76885223	0.00000000	0.00001829	-0.00000003
1269713375.00000000	1900.00000000	338.97949219	301.26004028	38.76909637	0.00000000	0.00026771	-0.00000004
1269713750.00000000	2000.00000000	338.97943115	301.26004028	38.76927185	0.00000000	0.00021816	-0.00000002
1269714000.00000000	2100.00000000	338.97943115	301.26004028	38.76933670	0.00000000	0.00026090	-0.00000001
1269714250.00000000	2200.00000000	338.97949219	301.26004028	38.76939392	-0.00000000	-0.00001274	-0.00000000
1269714625.00000000	2300.00000000	338.97949219	301.26004028	38.76944351	0.00000000	0.00004410	-0.00000000
1269714875.00000000	2400.00000000	338.97949219	301.26004028	38.76948166	0.00000000	0.00004049	-0.00000000
1269715250.00000000	2500.00000000	338.97949219	301.26004028	38.76950073	0.00000000	0.00009921	-0.00000000

forward_speed	pitch	roll	yaw	collision_vehicles	collision_pedestrians	collision_other	intersection_otherlane
0.00000000	0.00000000	0.00000000	-90.00029755	0.00000000	0.00000000	0.00000000	0.00000000
0.00000000	0.00000000	0.00000000	-90.00029755	0.00000000	0.00000000	0.00000000	0.00000000
0.00000000	0.00000000	0.00000000	-90.00029755	0.00000000	0.00000000	0.00000000	0.00000000
0.00000000	0.00000000	0.00000000	-90.00029755	0.00000000	0.00000000	0.00000000	0.00000000
0.00000000	0.00000000	0.00000000	-90.00029755	0.00000000	0.00000000	0.00000000	0.00000000
-0.00002576	0.00032102	0.00000000	-90.00029755	0.00000000	0.00000000	0.00000000	0.00000000
-0.01736914	0.05110347	-0.03173828	-90.00023651	0.00000000	0.00000000	0.00000000	0.00000000
-0.00053455	0.16052310	-0.14367674	-90.00042725	0.00000000	0.00000000	0.00000000	0.00000000
0.00276420	0.13918559	-0.13833623	-90.00038147	0.00000000	0.00000000	0.00000000	0.00000000
0.00253833	0.09959098	-0.10668945	-90.00021362	0.00000000	0.00000000	0.00000000	0.00000000
0.00179322	0.06689487	-0.07522581	-90.00010681	0.00000000	0.00000000	0.00000000	0.00000000
0.00122610	0.04393177	-0.05105591	-90.00006104	0.00000000	0.00000000	0.00000000	0.00000000
0.00077109	0.02862532	-0.03375243	-90.00000763	0.00000000	0.00000000	0.00000000	0.00000000
0.00048182	0.01869423	-0.02175904	-90.00006104	0.00000000	0.00000000	0.00000000	0.00000000
0.00032261	0.01232166	-0.01406861	-90.00005341	0.00000000	0.00000000	0.00000000	0.00000000
0.00020538	0.00823721	-0.00903320	-90.00005341	0.00000000	0.00000000	0.00000000	0.00000000
0.00009738	0.00679604	-0.00717163	-90.00005341	0.00000000	0.00000000	0.00000000	0.00000000
0.00009555	0.00510215	-0.00518799	-90.00005341	0.00000000	0.00000000	0.00000000	0.00000000
0.00006878	0.00374294	-0.00360107	-90.00005341	0.00000000	0.00000000	0.00000000	0.00000000
0.00004697	0.00278672	-0.00244141	-90.00005341	0.00000000	0.00000000	0.00000000	0.00000000
0.00002088	0.00238374	-0.00198364	-90.00005341	0.00000000	0.00000000	0.00000000	0.00000000
0.00002215	0.00198075	-0.00146484	-90.00005341	0.00000000	0.00000000	0.00000000	0.00000000
0.00001774	0.00166657	-0.00100708	-90.00005341	0.00000000	0.00000000	0.00000000	0.00000000
0.00001369	0.00143434	-0.00067139	-90.00005341	0.00000000	0.00000000	0.00000000	0.00000000
0.00000377	0.00133189	-0.00051880	-90.00005341	0.00000000	0.00000000	0.00000000	0.00000000

intersection_offroad	**steer**	**throttle**	**brake**	**handbrake**	**reverse**
0.00000000	-0.00023117	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023117	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023117	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023117	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023117	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023117	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023204	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00022932	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00022997	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023226	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023390	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023455	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023520	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023455	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023466	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023466	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023466	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023466	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023466	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023466	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023466	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023466	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023466	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023466	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023466	1.00000000	0.00000000	0.00000000	0.00000000
0.00000000	-0.00023466	1.00000000	0.00000000	0.00000000	0.00000000