NTNU
Norwegian University of
Science and Technology

End-to-End Steering Angle Prediction for Autonomous Vehicles

Anna Kastet
Ragnhild Cecilie Neset

Master of Science in Computer Science
Submission date: June 2018
Supervisor: Frank Lindseth, IDI
Co-supervisor: Faouzi Alaya Cheikh, IDI
Mahdi Amani, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

In recent years, convolutional neural networks (CNNs) have been applied to several autonomous driving tasks in an end-to-end manner with promising results. As these kinds of systems are given the freedom to self-optimise to maximise their overall performance, they are believed to eventually reach better performance than more fragmented solutions, where intermediate, human-selected criteria are optimised instead. End-to-end systems drastically reduce the amount of manual labour needed to annotate datasets.

This thesis explores end-to-end learning for autonomous driving using a SPURV Research robot. A CNN architecture trained on self-collected data is used to predict steering angles for three respective tasks: Indoor lane following using the SPURV robot, lane following inside the Udacity Simulator, and finally, a task in which a U-turn manoeuvre, which requires a notion of history, is performed. The goal is to check if this can be done with only feed-forward neural networks. The models for the first and last tasks are tested in real-life on the SPURV robot.

The indoor lane following was successful, and the SPURV robot followed lanes correctly and stably. The marker turning model showed less promising results: The SPURV was able to complete the task occasionally but with several failures. The SPURV vehicle was found to be a useful framework for data collection and testing of methods for autonomous driving and revealed that the quantitative metrics used in this project did not accurately reflect real-life performance.

Samandrag

Dei siste åra, har konvolusjonelle nevrale nettverk (CNN) vorte nytta til å predikere styringskommandoar i autonome bilar ved hjelp av ende-til-ende-læring med lovande resultat. Sidan slike system har eigenskapen at dei kan optimalisere seg sjølve for å maksimere systemet sin totale prestasjon, blir det anteke at dei snart vil prestere betre enn meir oppdelte system, der delmål definerte av menneske blir optimaliserte kvar for seg. Ein annan stor fordel med ende-til-ende-system er at dei reduserer mengda manuelt arbeid som må til for å annotere datasett.

I denne oppgåva blir ende-til-ende-læring for autonome bilar undersøkt ved å nytte ein SPURV Research-robot. Ein CNN arkitektur vert trena på sjølvsamla data for å predikere styringsvinklar for tre respektive oppgåver: Følgjing av ei innandørs køyrebane, samt køyring på ei bane i Udacity-simulatoren. I den siste oppgåva blir ein U-sving utført rundt ei kjegle, ei handling som krev ei form for historie i den nevrale modellen. Målet er å sjekke om ein så komplisert manøver kan lærast av eit nevralt nettverk utan sykliske sambindingar. Den første og siste oppgåva vert testa i verkelege omgivnader med SPURV-roboten.

Den innandørs banekøyringa var vellykka, og SPURV-roboten klarte fint å følgje banene riktig og stabilt. U-sving-oppgåva viste mangelfulle resultat: SPURV-roboten makta å fullføre oppgåva innimellom, men feila fleire gonger. SPURV-roboten viste seg å vere eit nyttig og effektivt rammeverk for innsamling av data og testing av metodar for autonom køyring, og avdekkja at dei kvantitative resultata i denne oppgåva ikkje alltid kunne reflektere modellane sine prestasjoner i røynda.

Preface

This thesis was written over the course of the spring semester of 2018 for the Department of Computer Science (IDI) at Norwegian University of Science and Technology (NTNU).

First of all, we would like to sincerely thank our supervisor Frank Lindseth for his inspiring and supportive guidance throughout the semester. Our gratitude also goes to Mahdi Amani, for providing us with exciting ideas and urging us to keep an eye on the details when we were tempted to take shortcuts. In addition, we thank Faouzi Alaya Cheikh for feedback on our drafts.

We would also like to thank Øyvind Kjerland, for helping us up on our feet at the beginning of the semester, and giving us helpful pointers to the project.

Many thanks to Thomas Frøysa at Revolve NTNU, for happily sharing his knowledge and resources with us.

For answering our questions quickly and even expanding the SPURV user guide at our request, we would like to thank the team at KVS Technologies.

Finally, we would like to thank Stein-Otto Svorstøl and Kristoffer Kjørlaug Larsen for technical help with Docker, and for moral support.

Ragnhild Cecilie Neset and Anna Kastet 06.06.2018

Contents

List of Figures

List of Tables

Glossary

backend A program not directly accessed by the user which performs a specialised function on behalf of a main system. 26

deconvolution A mathematical process to reverse the effects of convolution. 9

feature engineering The process of using domain knowledge of a problem to create features that make machine learning algorithms work. 1, 14

feed-forward neural network A neural network in which connections within the network do not form a cycle. i, 2, 6, 86, 97, 98

ground truth What is measured as the target value in the training and testing data for a machine learning system. 21, 46, 60–62, 67, 71, 72, 85, 86, 88, 90

hyperparameter In machine learning, a parameter whose value is set before training begins. 10, 12, 59, 78, 86, 93, 94, 98

moulding A strip of wood used as decorative architectural feature, often between a wall and the floor. 87

proxy task A task that a machine learning system is assigned to solve in addition to its main task. 18, 99

qualitative Descriptions or distinctions based on some quality or characteristic instead of some quantity or measured value. 36, 41, 59, 65, 69, 97

quantitative Measurable descriptions obtained through a quantitative process. Opposite of qualitative. 41, 59, 65, 69, 97

spatial Relating to space. 2, 7, 8, 17, 18, 33, 52, 53, 55, 56, 58, 72–77, 89–92, 98

temporal Relating to time. 18

tensor A generalisation of scalars, vectors and matrices to potentially higher dimensions. Can be viewed as N-dimensional arrays. 25

Acronyms

ALVINN Autonomous Land Vehicle In a Neural Network. 14

ANN artificial neural network. 2, 5, 6, 8, 22, 87, 96, 99

API application programming interface. 25, 98

BGR (blue, green, red). 36, 37

C-LSTM Convolutional Long-Short Term Memory. 18

CNN convolutional neural network. i, iii, 7–9, 11, 15, 16, 18, 22, 29, 47, 86, 92

CPU central processing unit. 26

CSV comma separated values. 26

CUDA Compute Unified Device Architecture. 25, 26, 33

cuDNN NVIDIA CUDA Deep Neural Network. 26

DAVE DARPA Autonomous Vehicle. 15

fps frames per second. 16, 36, 51, 55, 76, 87

GPS Global Positioning System. 21

GPU graphics processing unit. xxvii, 1, 21, 22, 25, 26, 29, 34, 64, 76, 87, 96

GTA-V Grand Theft Auto V. 19

HSV (hue, saturation, value). 38, 55, 80, 81, 91, 94

I/O Input/Output. 32, 34

LIDAR light detection and ranging. 16

LSTM long short-term memory. 16, 18, 99

MAE mean absolute error. 19, 35, 36, 58–60, 65, 67, 69, 70, 85, 88

MSE mean squared error. 7, 16, 35, 36, 58–60, 65, 67, 69, 70, 85, 88, 89

NTNU Norwegian University of Science and Technology. v, xxvii, 1, 2, 22, 95

PROMETHEUS Programme for a European Traffic with Highest Efficiency and Unprecedented Safety. 13

ReLU rectified linear unit. 6

RMSE root mean square error. 18, 20

RNN recurrent neural network. 6, 18, 86, 99

ROS Robot Operating System. 22–25, 30–34

SGD stochastic gradient descent. 11, 12, 35

VBP VisualBackProp. 9, 10, 16, 36, 41, 42, 47, 58, 61, 63, 68, 72–75, 78–81, 86–90, 92–94, 97, 98

Chapter 1

Introduction

This chapter introduces the thesis by giving a short summary of the background and motivation behind it. Then, a set of research questions are defined. The contributions of the thesis are mentioned, before introducing the chapter structure of the thesis.

1.1 Background and Motivation

In recent years, the interest in end-to-end learning for autonomous vehicles has increased. End-to-end learning denotes machine learning solutions where the system automatically learns internal representations of the task and outputs the next action to be made. The systems are never explicitly taught what features to look for. Hence, these approaches allow researchers to focus less effort on feature engineering and dataset annotation. It is believed that such systems will eventually reach better performance than more fragmented autonomous vehicle approaches, as the internal components can self-optimise to maximise overall system performance instead of optimising intermediate human-selected criteria. Simultaneously, the nature of end-to-end approaches makes these systems hard to analyse and reason about, as these intermediate human-selected criteria are no longer present.

NTNU recently acquired a SPURV Research vehicle from KVS Technologies[1]. The SPURV Research is a small car that is equipped with high-resolution cameras and a powerful graphics processing unit (GPU). Its compact size and technical specifications together with its indoor and outdoor capabilities make it suitable for research on autonomous vehicles. Large portions of this thesis are concerned with the utilisation of the SPURV in research on end-to-end learning in autonomous vehicles.

[1] `www.kvstech.no`

1.2 Research Questions and Objectives

This section summarises the goal of the thesis into two research questions. The overall goal of this thesis is to examine the possibilities for end-to-end learning of autonomous vehicles, especially targeted for the SPURV car. Thus, the following research questions have been formulated:

RQ1: How can an end-to-end lane following system for an autonomous vehicle (simulated or real like the SPURV robot) be trained and evaluated?

RQ2: Can an end-to-end feed-forward neural network learn to perform complex manoeuvres without any additional training information apart from visual cues and steering commands?

1.3 Contributions

This thesis is concerned with end-to-end learning for autonomous vehicles through the use of artificial neural networks (ANNs), and covers both the deep learning part as well as the utilisation of the SPURV Research car.

A model is trained on an end-to-end indoors lane following task, solving it with good results. Real-life tests using the SPURV car show that the model is able to follow indoor tracks with no problems. It is also able to correct its position when put in erroneous situations.

An additional task of doing U-turns around markers by using spatial history is solved with relatively unstable results. Real-life tests are partially successful, but not conclusive.

As this is the first attempt at using the SPURV for research purposes at NTNU, an essential part of this thesis has been understanding and setting up the SPURV system for autonomous vehicle research. The work has resulted in a comprehensive user guide to the SPURV. A variety of limitations, challenges and advantages of the system have been identified, making it likely that the findings in this thesis will facilitate future research projects using the SPURV at NTNU.

1.4 Thesis Outline

Table 1.1 presents an overview of the chapter structure of this thesis.

Table 1.1: An outline of the chapters and their contents in this thesis.

Chapter / Appendix	Description
1. Introduction	Introduction, motivation and research questions are presented.
2. Background	Theoretical background and related work relevant to the methods used this thesis are presented.
3. Methodology	Details of the methods used in the conducted research are presented.
4. Results	Results of the experiments are presented.
5. Discussion	Results from Chapter 4 are compared and discussed.
6. Conclusion and Future Work	A short conclusion based on the results and the following discussion is drawn in the context of the research questions. Pointers to future work are then presented based on the discussion chapter.
A. SPURV	Technical specification and comprehensive user guides for the SPURV related to data collection and end-to-end learning.
B. Code and Scripts	Summary of the relevant scripts used throughout this thesis.

Chapter 2

Background

This chapter covers the theoretical background and history related to this thesis. The basic theory of deep learning is summarised in Section 2.1. Section 2.2 covers the brief history of autonomous vehicles and the related work in end-to-end learning. Important hardware and software aspects related to the SPURV robot and training environment are then summarised in Section 2.3 and Section 2.4.

2.1 Artificial Neural Networks

An ANN is a model which takes a number of inputs and produces some kind of output. The input can, for instance, be the pixel values of an image, and the output a prediction of what object the image depicts. Deep learning is a method for dealing with ANNs with multiple layers. The book *Deep Learning* by Goodfellow et al. (2016), contains a thorough explanation on deep ANNs on which this section is based unless otherwise stated.

The basic unit of a neural network is a *node*, or *artificial neuron*, that is connected to other nodes by weighted links. The *weights* determine the strength of the link (Russel et al., 2014). An illustration of a single node can be seen in Figure 2.1. The illustrated node has three inputs a_1, a_2 and a_3. The weighted sum $x = \sum_{i=1}^{3} w_i \times a_i$ of these inputs is then passed to an *activation function* $f(x)$, which produces the output y of the node. The output is then passed on to subsequent nodes in the network, if any.

As ANNs usually consist of a large numbers of nodes, it is rare to look at individual nodes when creating and analysing the networks. Instead, the networks are viewed as a collection of layers, where the input values are sent into the input layer, and propagated to the output layer. In between, there's a number of hidden layers. ANNs

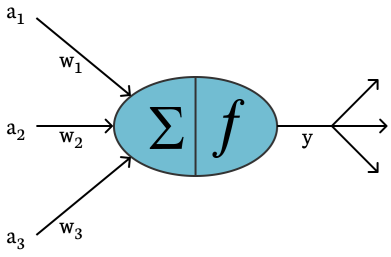


Figure 2.1: A single node in a neural network.

that only propagate the outputs to downstream layers, i.e. only in one direction through the network, are known as feed-forward neural networks. The opposite type of network is a recurrent neural network (RNN), where some outputs can be propagated to nodes or layers upstream, which means that the network can input some internal state to the next iterations.

ANNs can make use of different activation functions. These can also vary between different layers in a network. Historically, the *sigmoid* function

$$f_{sigmoid}(x) = \frac{e^x}{e^x + 1}$$

has been the most widely used. Unfortunately, when applied to modern networks with a large number of layers, the nature of the sigmoid causes the partial derivatives to diminish. Thus, the rectified linear unit (ReLU) function

$$f_{ReLU}(x) = max(0, x)$$

has become the default recommendation in modern neural networks. The tanh activation function is also a popular choice:

$$f_{\tanh}(x) = \tanh(x)$$

2.1.1 Training

A neural network learns by adapting its weights during training. In the case of supervised learning, the network is trained in iterations on labelled training examples with inputs $x = (x_1, ..., x_n)$ and corresponding target outputs $t = (t_1, ..., t_n)$. The collection of these training examples is called the *training set*.

When training examples are passed through the network, each weight w_i in the network is updated by

$$w_i' = w_i - \eta \frac{\partial C}{\partial w_i}, \tag{2.1}$$

where C is some cost function and w_i' is the new weight value. η is the *learning rate*, which scales the weight update. A cost function represents how well the network is

doing by calculating the difference between the network's output values y and the target values t for a given example. A commonly used cost function is the mean squared error (MSE)

$$C(t, y)_{MSE} = \frac{1}{n} \sum_{i=1}^{n} (t_i - y_i)^2, \tag{2.2}$$

where n is the number of elements in y and t. The choice of cost function is made based on the particular problem the network is designed to solve, *i.e.* MSE may not be suitable for every problem.

The gradients $\frac{\partial C}{\partial w}$ in Equation 2.1 are obtained for each weight in the network using the gradient descent algorithm, which utilises backpropagation and the chain rule to propagate the cost C, or error, upstream through the neural network.

2.1.2 Convolutional Neural Networks

A CNN is a specialised kind of neural network for processing data with a grid-like topology, like two-dimensional images. Unlike the kind of network previously presented, CNNs are able to utilise spatial information. The structure of a CNN was first proposed by Yann LeCun et al. (1998). Usages of CNNs have been very successful in practical applications.

A convolutional layer may be viewed as a collection of weights acting as a filter kernel. The kernel is applied in a systematic way to the pixels of an image. Figure 2.2 shows how the output from this operation is calculated, where each value within the kernel is a weight while each input value is the output from a node in a previous layer. The output from a convolutional layer is often called a *feature map*. A CNN usually consists of several such convolutional layers in sequence, with each new layer extracting higher-level features than the previous layer.

Images have properties that are insensitive to translation. A picture's overall meaning does not change if it is shifted one pixel to the right. CNNs take these properties into account by sharing the same weights across multiple image locations. The sharing of weights also lowers the number of unique parameters in the network, allowing for expansion of the network without requiring the corresponding increase in training data.

Modern CNNs are typically built using alternating convolution- and *max-pooling* layers. Max pooling, illustrated in Figure 2.3, outputs the max value of its input within a kernel. Although the alternating convolution and max pooling architectures are widely used, some research indicates that max pooling layers can simply be replaced by a convolutional layer with increased stride, without a noticeable decrease in performance in a variety of benchmark cases (Springenberg et al., 2014).

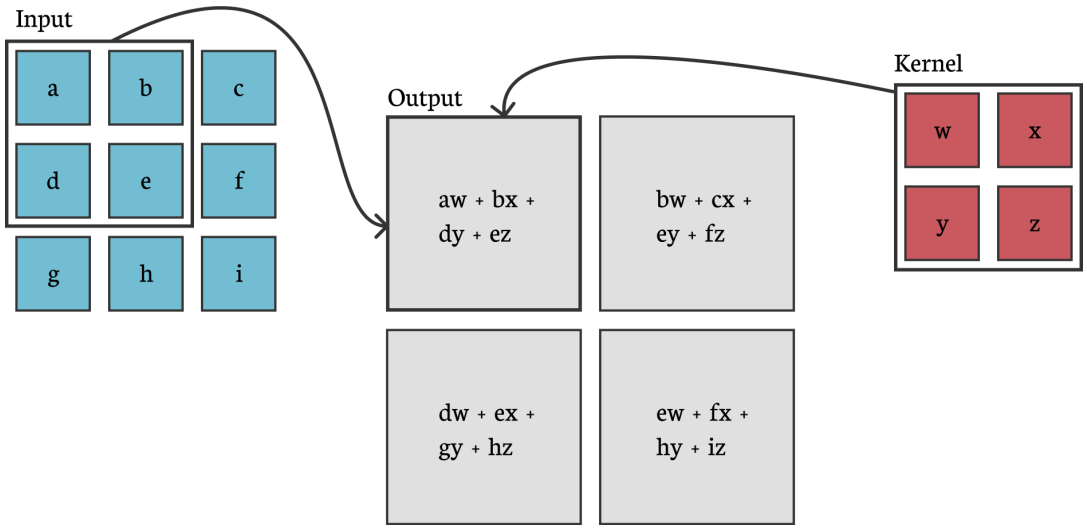


Figure 2.2: An example of convolution in 2 dimensions using a 2x2 kernel. The arrows indicate how the upper left corner of the output is created from combining the upper left corner of the 2-D input and the kernel.

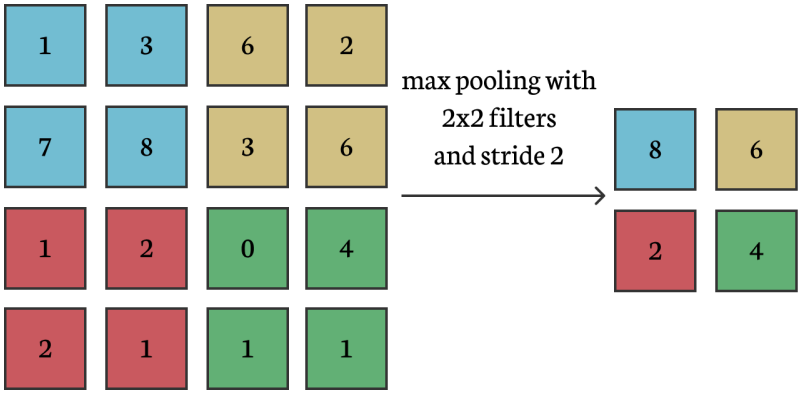


Figure 2.3: An illustration of max pooling with 2x2 filters and 2x2 stride.

2.1.3 Visualisation

In order to gain a deeper understanding of the inner workings of ANNs, numerous techniques have been developed to visualise neuron activation in the convolutional layers of CNNs. These techniques are often gradient-based, like saliency maps, or value-based, like Visual Backpropagation (Bojarski, Choromanska, et al., 2016).

Class Saliency Maps

Saliency maps display the spatial support of a given output class in a given image. This is done by computing the gradient of the output with respect to the image. This provides information about how the output will change with respect to small changes in the pixels. Saliency maps were first introduced in the paper by Simonyan

et al. (2013), where a saliency map for an image is created for a given class.

One of the shortcomings of gradient-based methods like the one by Simonyan et al. (2013) is that it only enables visualisations of a single class activation in a single layer. It thus works adequately for classification networks but cuts short when the network is used for a regression problem.

Visual Backpropagation

VisualBackProp (VBP) is a method proposed by Bojarski, Choromanska, et al. (2016), where the goal is to visualise which sets of pixels contribute most to the predictions made by the CNNs in order to verify that the predictions are based on reasonable optical cues. The method was originally developed for debugging and validation of an autonomous car system (Bojarski, Del Testa, et al., 2016), but transfers to any application using images and CNNs. Instead of calculating the gradients like Simonyan et al. (2013), VBP propagates the output values from a forward pass upstream through the network until a mask of the same size as the input image is obtained. These masks show the pixels that are responsible for the highest activation values in the convolutional layers.

Up-scaling of feature maps by using deconvolution is an important part of VBP. For each convolutional layer, the feature maps are averaged, resulting in a single feature map for every layer. Then, the averaged feature map of the deepest convolutional layer is scaled up to the size of the feature map of the previous layer. This scaled feature map is then point-wise multiplied with the previous layer's feature map. The resulting feature map is again scaled up to the size of the feature map of the layer before that one. This procedure continues up to the network's input layer, as illustrated in Figure 2.4, resulting in a mask of the same size as the original input image.

2.1.4 Regularisation

The ability to perform well on previously unobserved inputs is called generalisation. This is a central challenge in machine learning. The usual practice is to divide the dataset into two: The training set the model is used to train on and the independent test set of unobserved inputs. The measure of error on the training set is called the training error. Likewise, the error on the test set is called the test error. In some cases, the test error increases in favour of the training error decreasing. This is called *overfitting*, meaning that the model fits itself too much to the training set and is thus not able to generalise well enough on unseen examples. There are many strategies that are explicitly designed to reduce the test error at the expense of increased training error. These techniques are known as regularisation, whereof a few are introduced in this subsection.

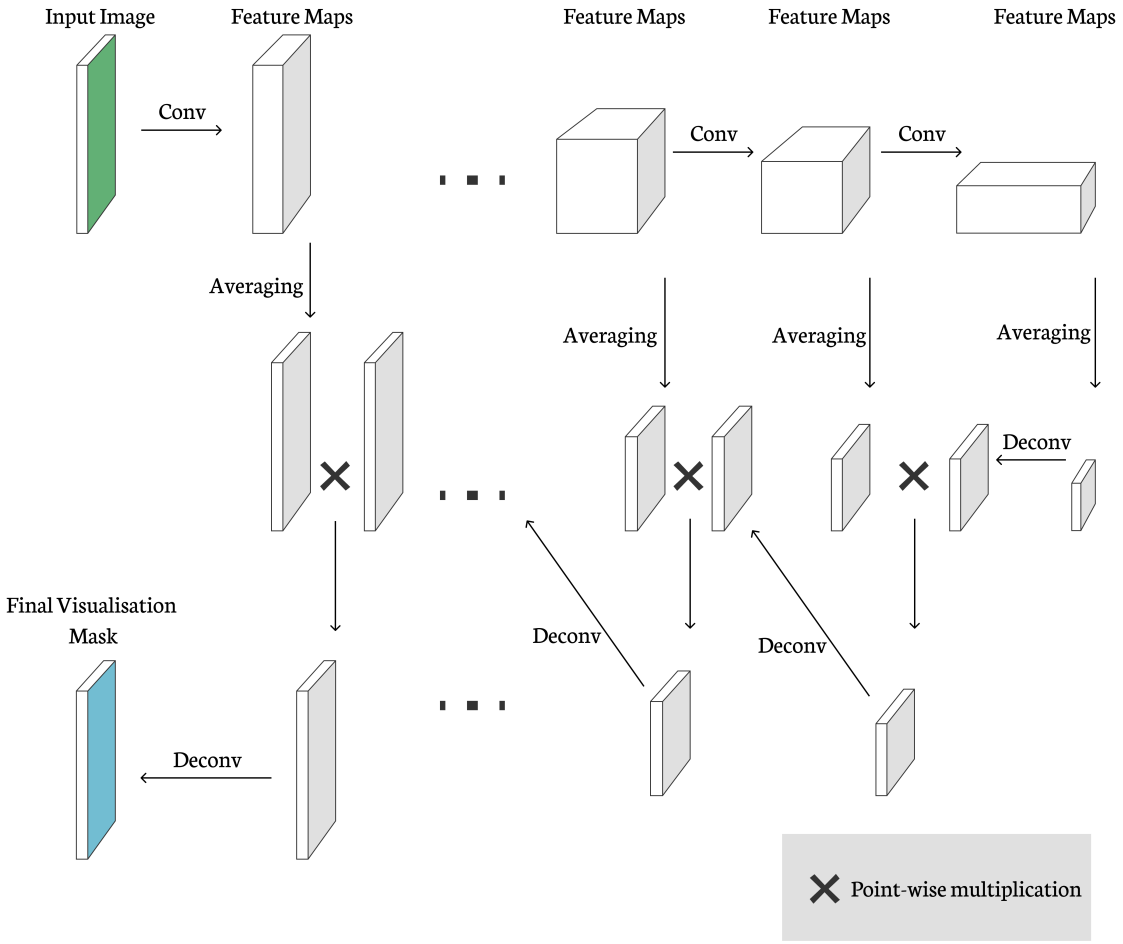


Figure 2.4: Illustration of the averaging, deconvolution and point-wise multiplication operations done on convolutional feature maps in VBP.

If the test error is actively used for generalisation, *information leakage* from the test set can occur, which causes the test error to be a biased estimation of the true error on unseen examples. In addition to the test set, a *validation set* is introduced, to be used for hyperparameter tuning during training.

Dataset Augmentation

The most straightforward way to make a machine learning model generalise better is to train it on more data. Unfortunately, the amount of data available is often limited. Creating synthetic data out of the data already available and adding it to the training set, is one method that often helps the model generalise.

Dataset augmentation has been proven to be particularly effective for various kinds of image applications. It is easy to simulate a wide range of variation in images.

Examples of augmentation techniques can be translating each image a few pixels in each direction, random rotation and horizontal or vertical flipping. While doing these operations, it is important that one makes sure the transformations do not make the object unrecognisable. For instance, extensive rotation or flipping can make an image of a traffic sign lose its intended meaning.

In *random erasing* by Zhong et al. (2017), a randomly selected rectangle region of an image is erased and replaced by random or constant values. This technique produces training images with various levels of occlusion.

Pre-trained Networks

When trained on images, the earliest layers in a CNN tend to learn features that are prevalent across different problems. It has become common practice to use the weights from CNNs pre-trained on a dataset like for instance ImageNet (Russakovsky et al., 2014) to solve different computer vision tasks. This approach is called *transfer learning*. It is also possible to start with a pre-trained network, add some final layers and then fine tune the entire network for some specific task. Both these approaches can improve generalisation performance, meaning that it can be a useful technique for improving the performance of deep neural networks (Yosinski et al., 2014).

Early Stopping

Another technique commonly used to avoid overfitting, is early stopping. During training, the performance on both the training and validation set is monitored. When the error on the validation set starts increasing in favour of the error on the training set decreasing, the training is stopped. The test set is used to evaluate the final performance of the trained model.

2.1.5 Optimisation

Optimisation algorithms are an important part of deep learning. Optimisation algorithms that process the entire training set simultaneously in one large batch, are called batch or deterministic gradient methods. Most algorithms used for deep learning use more than one but fewer than all training examples in each batch when training, so-called minibatches. These methods are commonly called stochastic methods. This section explains the three stochastic optimisation algorithms stochastic gradient descent (SGD), RMSProp and Adam.

Stochastic Gradient Descent

SGD simply applies the updates to the weights by computing the average gradient for each minibatch. In practice, it is necessary to decay the learning rate with time when using SGD, because the random sampling of each minibatch, unlike batch methods, adds noise that does not vanish.

During training, learning can be slow because if the gradient is small, the next weight update is also small. An improvement can be to add momentum to the update rule. Momentum is designed to speed up learning. The momentum algorithm accumulates a decaying moving average of past gradients and continues moving in their direction. This can help to get out of local minima.

RMSProp

RMSProp is one of several algorithms with adaptive learning rates, where every parameter has a separate learning rate that is adapted throughout the course of learning. The learning rates in RMSProp are adapted by using an exponentially decaying average of the gradients for each weight, meaning that it discards history from the extreme past.

Adam

Adam can be viewed as RMSProp with momentum, with a few distinctions. The momentum in Adam is directly incorporated as an estimate of the first-order moment of the gradient. Adam also takes advantage of the second-order moments of the gradients in addition to the first-order moments to adapt the parameter learning rates. Adam is generally regarded as a robust optimisation algorithm when it comes to the choice of hyperparameters, although the learning rate sometimes must be tuned.

2.2 End-to-End Learning for Autonomous Vehicles

This section summarises the history and advances of autonomous vehicles, focusing on the end-to-end learning approach. Issues concerning path planning, inter-vehicle communication and other aspects not related to autonomous perception are not covered. Section 2.2.1 introduces the more traditional mediated perception approaches in the field, while different aspects of end-to-end learning are summarised in Section 2.2.2 and Section 2.2.3.

2.2.1 Mediated Perception Approaches

Mediated perception approaches are defined by C. Chen et al. (2015) as approaches that involve separate components, where each component is responsible for a subset of the aspects relevant for driving. Some of the aspects can, for instance, be related to image recognition, like detection of lanes, traffic signs or pedestrians. These results are then combined to form an environment model of the vehicle's surroundings. This environment model is fed to a decision system that resolves the upcoming action to be performed by the vehicle.

Most autonomous driving systems from the industry incorporate mediated perception approaches. In the 1980s, numerous projects on autonomous driving were started by governmental institutions worldwide. In 1986 the Programme for a European Traffic with Highest Efficiency and Unprecedented Safety (PROMETHEUS) project was initiated in Europe. The project involved a large number of vehicle manufacturers, universities and research units from governments of 19 European countries. One of the most notable achievements in the early stages of the PROMETHEUS project were the ones of Dickmanns et al. (1994). The VaMORs-P vehicle performed the first autonomous drive in 1995 from Munich in Germany to Odense in Denmark with about 95% of the driving being autonomous. The VaMORs-P system architecture is structured in an object-oriented manner, with distinct components being responsible for clear tasks such as detection of traffic signs, moving humans and other objects. The output from these components is saved in a database, where other distinct components resolve the vehicle control actions.

During the past years, the interest in autonomous vehicles has increased dramatically, both in academia and the industry. Companies like Tesla, Google, Uber and Toyota are all testing highly autonomous cars on real roads, while companies like Ford, Audi, BMW, Nissan, Volvo, Bosch and Daimler-Benz have invested heavily in research and are in the process of launching driving assistants of different levels of intelligence (Chan, 2017). Baidu, BMW, Ford and Volkswagen further promise to have fleets of autonomous vehicles in production by 2021. Chan (2017) further notes that the exact level of autonomy that is achieved or that is planned to be achieved by these companies is, more or less intentionally, imprecisely described. Issues are assumed to persist in most of these highly autonomous systems, for instance, high reliance on internal pre-mapping of roads, which causes an inability to drive in unknown locations.

Advantages and Limitations

Autonomous driving systems using mediated perception approaches have the advantage of being easier to reason about. The clear distinction between the different components makes testing of each component simpler, making it possible to examine how the vehicle would act in various situations by investigating the details of the

components' designs.

A known problem with mediated perception approaches is that the information from the different components often needs to be converted to the same format before creating the environment model. This conversion can result in additional noise, which degrades the accuracy. Another disadvantage of these approaches is that they often require labelled training data or the utilisation of careful feature engineering in order to work properly. These tasks are time-consuming as large amounts of manual human work is required. The high degree of detailed human engineering of the different components in the system also increases the likelihood of missing important design details (C. Chen et al., 2015).

2.2.2 End-to-End Learning

This section starts with a brief history of end-to-end learning efforts in autonomous vehicles. Approaches of particular interest for this thesis are then covered.

In the context of autonomous vehicles, the term end-to-end learning is also known as *behaviour cloning* or *behaviour reflex approaches*, as proposed by C. Chen et al. (2015). The phrase denotes an approach in which a single model learns to predict steering commands directly from the sensor input. The end-to-end approach results in a very simple design, where the different components of a typical mediated self-driving car, including image recognition, path planning and decision-making, are all replaced by one neural network, as shown in Figure 2.5. Since the different components do not have to be designed and optimised separately and with human intervention, the manual effort required to set up a running solution is arguably much smaller than for a mediated perception system. In exchange, understanding and controlling the decision-making process, which is a crucial aspect of developing fully autonomous vehicles, is much more difficult.

An end-to-end model can be trained using recordings of human driving. Training data typically consists of images from a forward facing camera or other sensors, that are paired with the measured steering action. The loss is then calculated from the difference in predicted and recorded steering commands during training. This method produces a more or less automatically labelled training set, although some categorisation of the recorded driving may be useful.

Early Attempts and PilotNet

Before 2016, few projects on end-to-end learning for the autonomous driving task existed. The first known project is the Autonomous Land Vehicle In a Neural Network (ALVINN) project (Pomerleau, 1989). A fully connected neural network was trained for doing lane following on a collection of simulated black and white

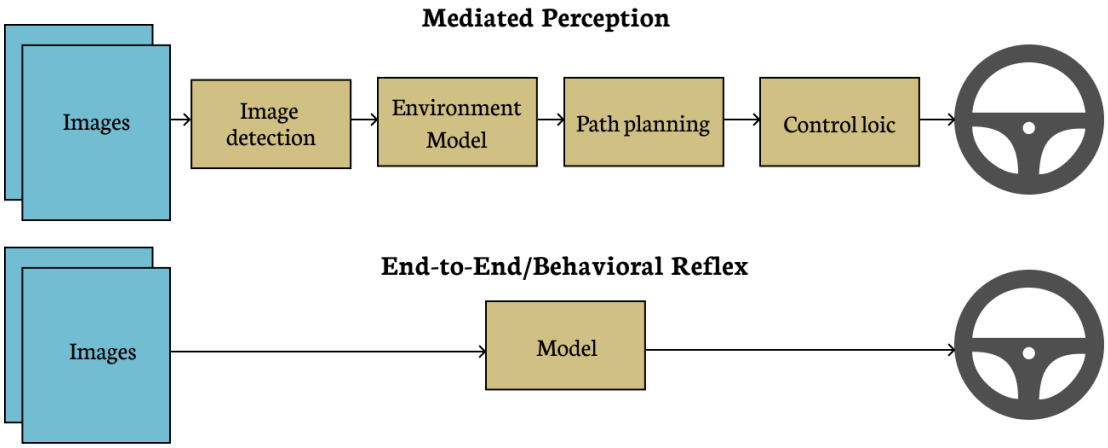


Figure 2.5: The difference between end-to-end and mediated perception to autonomous driving with camera input.

road images. In 2006, the DARPA Autonomous Vehicle (DAVE) project (LeCun et al., 2006) explored end-to-end learning further, this time upgrading to a CNN and a stereo camera. DAVE was trained for doing obstacle avoidance in a natural outdoor environment. Both projects got fairly good results and proved that it was possible to apply end-to-end learning for autonomous driving tasks.

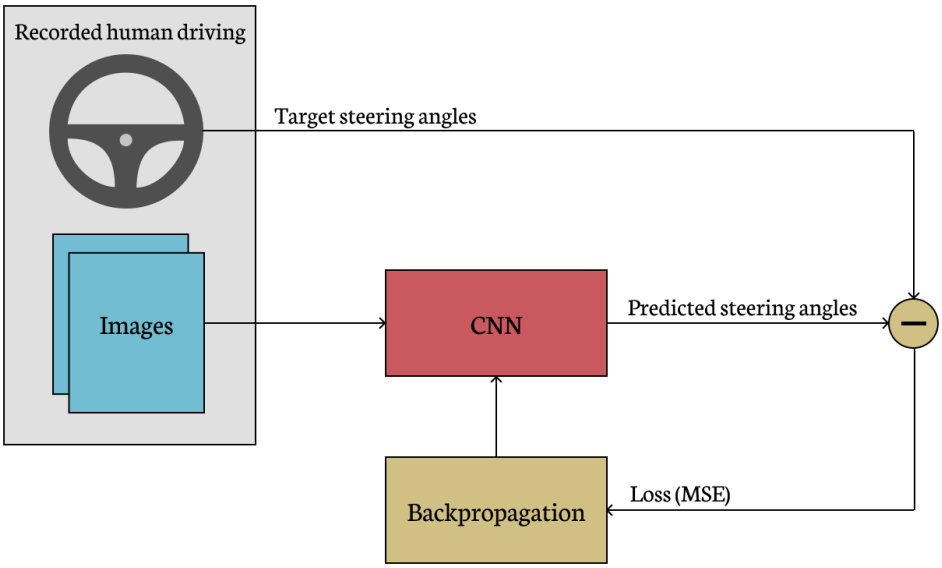


Figure 2.6: An overview of the PilotNet system.

In 2016, the DAVE-2 (Bojarski, Del Testa, et al., 2016) project, also known as *PilotNet* (Bojarski, Yeres, et al., 2017), was started to further explore and develop end-to-end learning. PilotNet was trained and tested for predicting wheel steering angles with the goal of following lanes on regular roads and highways. An overview

of the system is shown in Figure 2.6. The raw training data consisted of around 70 hours of driving, recorded from two different vehicles. Everything irrelevant for lane following, like lane changing and turning into new roads, was filtered out of the training data. Two additional, side-facing cameras, one on each side of the car, was used to gather data from incorrect positions. Rotations and shifts of the three viewpoints and the corresponding steering angle were then applied to the training data to simulate examples on regaining correct positioning when departed from the middle of the lane. PilotNet consisted of one normalisation layer, five convolutional layers and three fully connected layers. It is assumed that image recognition resulting in some form of environment model is done in the convolutional layers and that the control logic happens in the fully connected layers, but this cannot be known for sure. The MSE loss function was used during training. The model was tested inside a driving simulation before being taken out for driving on real roads. The result was a system that successfully followed various roads and lanes with very few errors, driving autonomously between 98% and 100% of the time (see Section 2.2.3 for an explanation of this metric). The system operated at 30 frames per second (fps). Visualisations of the activation in the convolutional layers, the VBP maps introduced in Section 2.1.3, showed that the network focused heavily on the outlines of the road, which means that the network had by itself learned to recognise the most important features for doing lane following.

The promising results of PilotNet caused a wave of interest in end-to-end learning, and since its publication, several articles and projects on the subject have emerged. Predicting steering angles for lane following with images as input appears to often serve as a basic problem in the field, from which comparisons and expansions emerge. The Udacity Challenge #2[1], released in 2016, was a challenge made by Udacity and Google for solving this exact problem, the purpose being to aid Udacity in making a car that could do what PilotNet did. Some of the articles that have been published focus on widening the scope of PilotNet by introducing for example navigation and obstacle avoidance (Hubschneider et al., 2017) or speed control (Yang et al., 2018). Others utilise new sensors like a light detection and ranging (LIDAR) (Caltagirone et al., 2017). Some articles present experiments with different classes and architectures of neural networks, like long short-term memory (LSTM) (Eraqi et al., 2017) or reinforcement learning (Sallab et al., 2016). These approaches are summarised in the rest of this section.

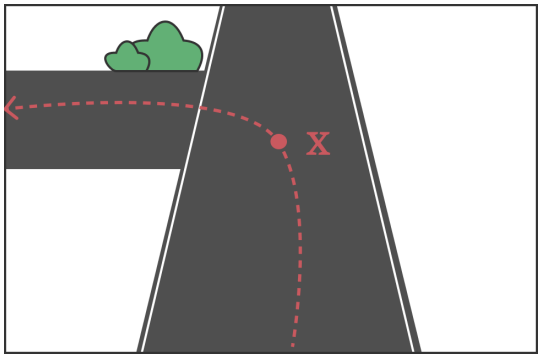
Relevant Approaches

In the setup of Bojarski, Del Testa, et al. (2016), a feed forward CNN is fed an image of the current front view from the vehicle from which it predicts the best steering command. This approach is very clean and simple but does have some limitations. The following sections cover some architectural setups that are found to be particularly interesting or relevant to this thesis.

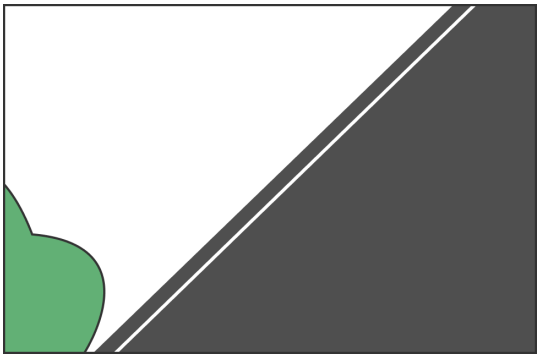
[1] www.medium.com/udacity/challenge-2-using-deep-learning-to-predict-steering-angles-f42004a36ff3

Navigation In the works of Hubschneider et al. (2017) a car was, in addition to lane following and obstacle avoidance, trained to follow simple navigational instructions. The navigational inputs were in the form of binary signals for making either a left turn (or lane change), a right turn (or lane change) or no turns. The signals appeared a short while before the junctions, and the model itself was in charge of deciding when to initiate the turn and for behaving correctly during the turn. The input signals skipped the convolutional layers and were fed directly into the fully connected part of the network, where the control logic is assumed to take place. The architecture of Hubschneider et al. (2017) was very similar to the one by Bojarski, Del Testa, et al. (2016), but with one additional fully connected layer after the convolutional layers. Making the car act on the input signals was challenging at first, but up-scaling the signals by a factor λ helped, and in the end, the car was able to follow the navigation instructions almost perfectly. This research shows that an end-to-end model is able to incorporate some purpose in its decision-making.

Spatial history Another interesting aspect of the works of Hubschneider et al. (2017) was the decision to use a spatial history of images at each prediction. Using spatial history simply means sending multiple images into the neural network simultaneously, where in addition to the most recent image, a given number of images from earlier spatial locations are included. The idea behind this approach was that the car would get a too limited camera view of its surroundings in one single moment to manage to do obstacle avoidance and turning. For example, most intersection turns are so steep that a regular camera would not be able to see the outline of the destination road, as illustrated in Figure 2.7. Two additional images, from 4 and 8 meters back were therefore fed into the network at each prediction. Compared to models without spatial history, the model with three images inputted was described to reduce errors significantly while increasing prediction time only slightly.



(a) The camera view from a car a few meters before a left turn. The dashed red line indicates the future path of the car



(b) The camera view from the car when it is in position x in Figure 2.7(a)

Figure 2.7: Illustrations of limited camera view during a left turn.

Temporal dependencies The goal of using spatial history is to use information from earlier states to give the model a better understanding of its current environment state. A similar approach is to use a RNN to handle temporal dependencies between image frames. Eraqi et al. (2017) propose the Convolutional Long-Short Term Memory (C-LSTM) architecture, which is a version of LSTM that is adapted to fit the task of autonomous driving using end-to-end learning. The first layers of the C-LSTM architecture are convolutional and the output of the final convolutional layer is a feature vector that is assumed to contain a form of environment model. The environment models of the current and past layers are then combined in a LSTM layer, which finally is fed into the fully connected layers that are assumed to do the control logic. Eraqi et al. (2017) use the C-LSTM to try to improve angle prediction precision on the Comma.ai dataset (Santana et al., 2016). The proposed C-LSTM architecture was able to improve the root mean square error (RMSE) and the whiteness by 35% and 87% respectively compared to a state-of-the-art CNN solution (see Section 2.2.3 for an explanation of these metrics).

Proxy supervision Cermak et al. (2017) focus on training a model for the task of stopping for pedestrians. Images from the KITTI dataset (Geiger et al., 2013) were automatically labelled as either "stop" or "go" cases, based on the occurrence of bounding boxes with pedestrians and their position. The model was then trained for predicting to correct action ("stop" or "go") from the image input. In order to force the car to recognise pedestrians, the model was trained for the additional, proxy task of predicting bounding boxes around any pedestrians. This technique was shown to increase the accuracy of the model, compared to pure end-to-end action prediction.

Virtual Environments

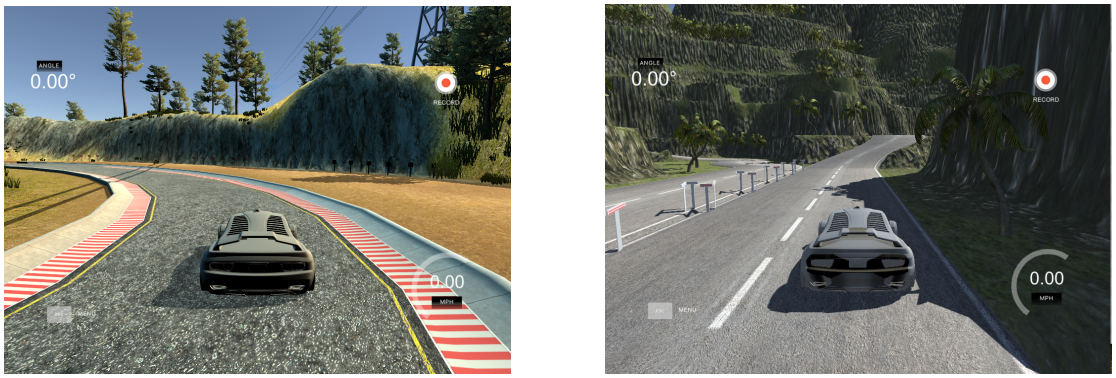


Figure 2.8: Two different tracks from Udacity's Self Driving Car Simulator.

Both gathering data and testing an autonomous vehicle is a time consuming and costly task that requires access to an actual vehicle and expensive additional equipment. For this reason, experimenting with virtual environments has become popular. The works of Martinez et al. (2017) found that using driving recordings from the game

Grand Theft Auto V (GTA-V) in the training and testing of autonomous cars may be a powerful tool, especially for pre-training or for expanding a real-life dataset. They point out that virtual environments make it possible to test the model on out-of-ordinary scenarios like near-accidents. Virtual environments are also useful for experimenting with augmentation, network architectures and other parameters.

The Udacity Self-Driving Car Simulator[2], hereby denoted as the *Udacity Simulator*, gives users an easy way to gather data and test driving systems inside a virtual road environment, as shown in Figure 2.8. The simulation is not as realistic as for instance GTA-V, but features two tracks with a different appearance. The simulator was made for students taking the Udacity Self-Driving Car Nanodegree course[3], but is open for everyone. Numerous projects on end-to-end learning using the Udacity Self-Driving Car Simulator are available online.

2.2.3 Evaluating Performance

There are few consistencies found between the metrics used when testing an end-to-end trained autonomous vehicle. This is because evaluating the performance of a model that predicts steering angles is generally challenging. One approach to is to do the static testing which is common for machine learning tasks, where recorded test data is sent into the model and used for frame by frame comparisons between prediction and target steering angles. A more suitable way of testing the model is to let it try to follow lanes in an interactive environment and observing its behaviour. This can either be done inside a simulation or in the real world. Methods and properties for both static and interactive approaches will be covered in the rest of this section.

Static Testing

Static frame by frame evaluation is simple but inaccurate. To begin with, following the exact middle of the lane at all times is practically impossible, so some amount of noise will be constantly present in the target data, resulting in low scores for typical regression metrics like mean absolute error (MAE). Moreover, there are numerous variations in how to steer a vehicle to follow a lane, so frame by frame comparison of steering angles between a target driving path and a predicted path may in many cases have a weak connection to on-road performance. Z. Chen et al. (2017) illustrates this with an example of a straight forward target driving path, and two very different model-predicted driving paths, shown in Figure 2.9. In the image to the left, the predicted path goes a little to the right and then straightens up, which is acceptable behaviour. The other path continues going to the right and exits the lane, which

[2] `www.github.com/udacity/self-driving-car-sim`

[3] `www.udacity.com/course/self-driving-car-engineer-nanodegree--nd013T`

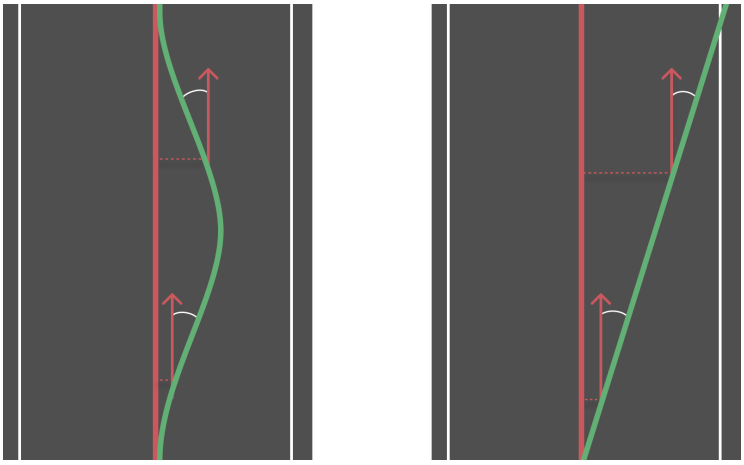


Figure 2.9: Examples of two model-predicted driving paths (green lines), compared to straight forward target path (red lines). The left driving path is acceptable, while the right path is erroneous. However, both paths will have approximately the same error compared to the straight forward path, as only the angles are compared.

is erroneous behaviour. However, both driving paths will have approximately the same measured error, as the mean divergence of the predicted steering angles from the target angle is approximately the same. Due to these reasons, deciding on some threshold value of a static error measure that is good enough to result in the desired behaviour is difficult. Static testing is still useful for getting an idea of the model's behaviour and for comparing the performance of different models.

Eraqi et al. (2017) use static frame by frame testing to compare the different model architectures proposed in the article. They argue that RMSE, as shown in Equation 2.3 is the best metric to use for angle predictions, because it punishes large deviations more than smaller ones. With $n = |t| = |y|$:

$$C(t, y)_{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(t_i - y_i)^2} \tag{2.3}$$

To overcome some of the limitations of the regular error measures, Eraqi et al. (2017) used whiteness, Equation 2.4, as an additional metric in their comparisons. Whiteness measures the smoothness of a signal, punishing sudden shifts in predictions, which would suggest that the model has unstable behaviour.

$$W = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\partial Y(\tau)}{\partial \tau}\bigg|_{\tau=i}\right)^2 \tag{2.4}$$

Interactive Testing

Bojarski, Del Testa, et al. (2016) made an evaluation metric that tries to capture the percentage of time that the car operates autonomously, which was used for evaluating both the simulation and real-world tests. This was done by giving a six-second punishment for every human intervention needed and dividing this by the total duration of the autonomous drive

$$\text{autonomy} = (1 - \frac{\text{number of interventions} \times 6 \text{ seconds}}{\text{elapsed time}}) \times 100. \tag{2.5}$$

Interventions were to be done if the car departed from the centre of the lane by more than one meter, and six seconds was used for punishment because this was the estimated time needed for a human to steer the car back into the correct position and then resume autonomous mode.

Hubschneider et al. (2017) measured the performance of their autonomous car on Global Positioning System (GPS) recorded driving paths. A ground truth driving path was found by averaging three different recorded human paths on a route. The autonomous car was then set to drive the same path, and the max and mean distance between this path and the ground truth driving path was calculated.

Arguably, the most accurate way of testing an autonomous car is to let it drive on actual roads. However, testing an autonomous car on real roads is dangerous, time-consuming and costly. Testing inside a simulation is, therefore, a good alternative, which is commonly used both as preliminary testing and as an alternative to real-world testing. The simulation technique used for testing PilotNet (Bojarski, Yeres, et al., 2017) uses a real-world driving recording. The images are first shifted to show the view from positions in the exact middle of the lane, to correct for possible human errors. For each frame, the new simulated position is calculated from the current simulated position and the predicted steering angle. The next image from the recording is then shifted to match the car's position, making the testing work in an interactive manner. The simulation keeps track of the car's deviation from the middle of the lane.

2.3 Hardware

This section summarises the hardware aspects of this thesis, starting with an introduction of the SPURV vehicle as a tool for data collection and autonomous vehicle research. Then, the importance of GPUs for deep learning is explained.

2.3.1 SPURV Research Vehicle

The SPURV Research by KVS Technologies[4], pictured in Figure 2.10, is a 50 x 45 x 25 cm electrical robot developed to provide a platform for research and development of autonomous vehicle technology. The version acquired by NTNU is equipped with two cameras: One in the front, and one in the rear end of the car. The SPURV is suitable for both inside and outside use and can reach a max speed of 45 km/h. An NVIDIA Tegra X2 GPU is installed as the onboard computer. Figure 2.11 shows an overview of the various hardware components in the car and how they are connected. The complete specification of the car can be found in Appendix A.1. The SPURV system is based on Robot Operating System (ROS), which is introduced in Section 2.4.1.

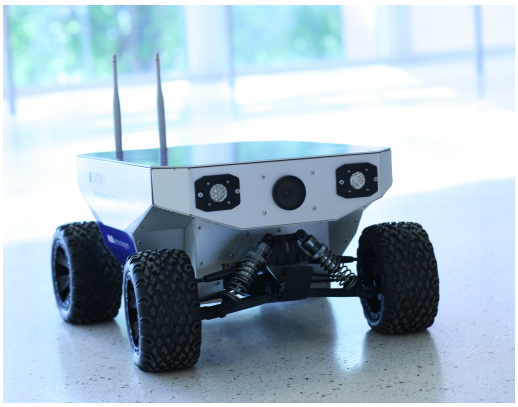


Figure 2.10: A photograph of the SPURV robot.

2.3.2 Computations Using GPUs

GPUs are processors designed to accelerate three-dimensional graphics that are displayed to users. Initially, this was a feature of high-end workstations, but since the late 1990s, they became more and more common in personal computers and gaming consoles. Due to the trends in computer gaming, the GPUs have been designed with increasing internal parallelism and internal possibilities for programming. The high level of parallelism in GPUs, has made the units attractive for solving other computationally intensive tasks unrelated to high-resolution computer gaming. ANNs can be represented as matrices, making it possible to do both training and prediction in a parallelised fashion. Chellapilla et al. (2006) pioneered the use of a GPU for training a deep CNN in 2006. Ever since, even more powerful GPUs have hit the market, making it possible to train deeper networks on more data in less time.

[4] `http://kvstech.no`

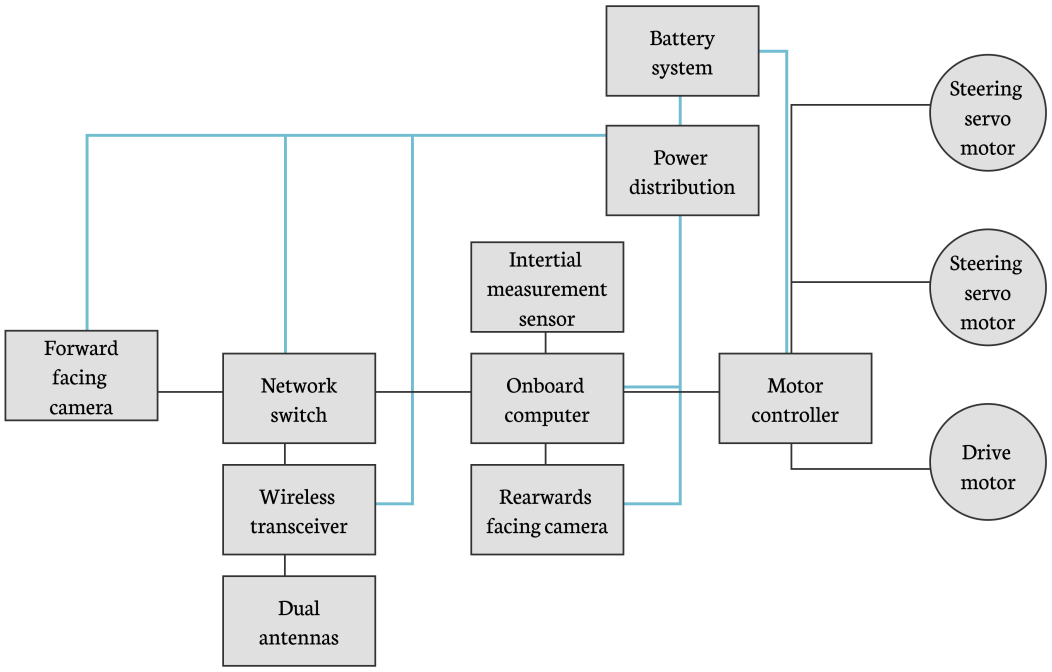


Figure 2.11: A block diagram of the hardware components in the SPURV car and how they are connected. The blue lines denote power distribution.

2.4 Software

This section summarises important software aspects in this thesis. Section 2.4.1 introduces the Robot Operating System (ROS), which is relevant for both data collection and control of the SPURV robot. In Section 2.4.2, the importance of CUDA is explained, while Section 2.4.3 introduces the deep learning libraries TensorFlow and Keras.

2.4.1 Robot Operating System

ROS[5] is a framework for writing robot software. With a collection of tools, libraries and conventions it aims to simplify development of complex robots across a wide variety of platforms. Several aspects of ROS have been important in the context of this thesis. These are presented in Figure 2.12 and briefly explained in this subsection. Unless otherwise stated, this explanation is based on the *ROS Wiki* (n.d.) and Quigley et al. (2015).

[5] `www.ros.org`

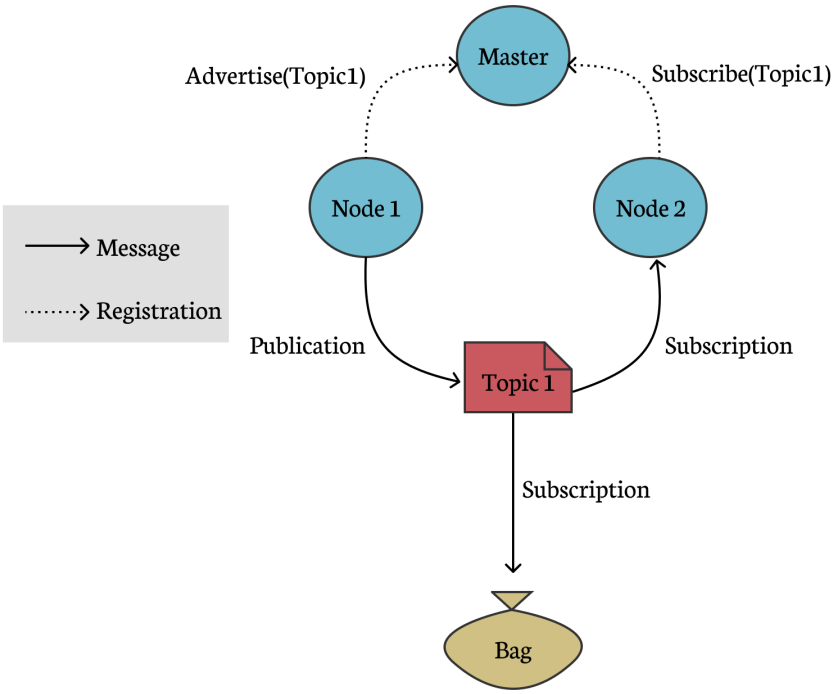


Figure 2.12: A simple figure of the relevant graph concepts of ROS and how they are connected through message passing. Node 1 first notifies the master that it wants to advertise to topic 1. Node 2 registers its subscription to topic 1. Any messages published to topic 1 will then be passed on to node 2. If a bag is recording topic 1, it will also receive these messages.

Nodes

ROS uses the concept of nodes. A node is a process that performs computations. A robot control system will usually comprise many nodes that are combined together into a graph. The nodes communicate with each other by using topics.

Topics

Topics are named buses over which nodes exchange messages. Topics decouple the production of information from its consumption by utilising the publish-subscribe pattern. Nodes publish information they produce to relevant topics. Similarly, nodes can subscribe to a topic if they are interested in it.

The messages published to topics are simple data structures consisting of typed fields. Standard primitive types such as integers, floating point numbers and booleans are supported. A message may include a header, which includes common metadata such as a timestamp.

Bags

Bags can be used to collect data from a robot. When a bag is recording, it can subscribe to one or more specified topics and store the messages as they are received. A bag can later be played back in ROS, making the bag publish the same messages it collected during recording in the same order as they were received.

Master

The ROS master provides naming and registration services to the rest of the nodes in the system. Its main role is thus to enable individual nodes to locate one another. ROS is designed to support distributed computing. Hence, one can run nodes distributed on multiple computers. As long as all nodes have access to the same master node in the network, they can communicate through topics as if they were in the same physical location.

In theory, a node should make no assumptions about where in the network it runs and should be able to be relocated to a different machine at any time. An exception to this are nodes that communicate directly with a piece of hardware. Such nodes must run on the same machine as the hardware is connected to.

2.4.2 CUDA

Compute Unified Device Architecture (CUDA)[6] is a parallel computing platform and application programming interface (API) for general purpose programming on NVIDIA's line of GPUs. It is a software layer that gives software engineers and developers direct access to the virtual instruction set of NVIDIA GPUs, making it unnecessary to implement custom interfaces for every application. The generality of CUDA has enabled the development of a number of tools for a wide variety of applications. One of these is TensorFlow, introduced in Section 2.4.3.

2.4.3 TensorFlow and Keras

TensorFlow[7] is a tensor library developed by Google that handles low-level calculations such as tensor manipulation and differentiation. All computations and states in TensorFlow are represented as a single dataflow graph. Each vertex in the graph represents an operation, and each edge represents the tensor from or to a vertex. As all communication between sub-computations is explicitly expressed, it is easy

[6] `www.developer.nvidia.com/cuda-zone`
[7] `www.tensorflow.org`

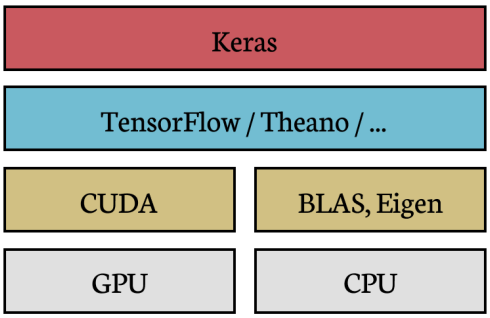


Figure 2.13: The deep learning software and hardware stack with TensorFlow and Keras.

to execute the computations in parallel and even partition them across multiple devices (Abadi et al., 2016). TensorFlow runs seamlessly on both GPUs and central processing units (CPUs). When running on a GPU, TensorFlow wraps the NVIDIA CUDA Deep Neural Network (cuDNN) library of optimised deep-learning operations (Chetlur et al., 2014), which uses CUDA for its basic operations.

The generality of TensorFlow makes its syntax relatively complex. In order to simplify development of deep learning models, one can use high-level libraries such as Keras[8]. Keras provides building blocks for developing deep learning models. It does not handle low-level calculations, but relies on a backend to do so. As Keras is designed in a modular way, it supports multiple libraries as its backend. This means that one can run Keras with any supported backend without having to make changes to the code. One of these backends is TensorFlow. Alternative backends for Keras are Theano[9] and Microsoft Cognitive Toolkit[10] (Chollet, 2017). The software and hardware stack using TensorFlow and Keras can be seen in Figure 2.13.

2.4.4 Udacity Simulator

The Udacity simulator, further introduced in Section 2.2.2, provides a simulated 3D environment with two different tracks on which a virtual car can drive. The simulator can both be used for collection of training data, and virtual testing of the trained model.

Collecting Data

A record-button inside the simulation can be pressed to automatically store data in a training-ready structure, which contains images from an imagined forward-facing camera and two side-facing cameras on the car, in addition to a comma separated

[8] `www.keras.io`
[9] `www.deeplearning.net/software/theano`
[10] `www.github.com/Microsoft/CNTK`

values (CSV) file containing steering angles and other parameters along with file paths to the images. Steering can be done using keyboard or mouse. The simulator is built using the game making engine Unity[11], in which the tracks can be modified. For instance, the road can be painted with a different texture or obstacles can be placed in the path.

Running the Model

It is possible to communicate with simulator through sockets. The simulator broadcasts all driving data to a specific port number and can receive and publish steering actions when the autonomous mode is chosen. Udacity provides a Python script that can be used to specifically test trained models that do steering angle prediction. The script can also record images and steering commands from the autonomous drive.

[11] `www.unity.com`

Chapter 3

Methodology

This chapter summarises the implementation and setup needed to perform the experiments in this thesis. First, the setup and configuration needed to use the SPURV for data collection and for autonomous driving are described briefly. Then, the various aspects of implementing the neural network are covered, including data preprocessing and augmentation, design and parameter choices. The implementations of the utilised visualisation techniques are explained, before each of the different learning tasks and the neural network architectures used are described in detail.

3.1 Brief Overview

An overview of the components used in this research project can be seen in Figure 3.1. An end-to-end CNN was set up to do steering angle prediction on different tasks. The SPURV was used for collecting data, and the data was exported and preprocessed. The models were trained on separate lab computers with GPUs. The finished models were then transferred back to the SPURV and tested for autonomous driving.

Three models were trained for solving three separate tasks. The first task was to do lane following on an indoor lane, simulated by laying two parallel ropes on the floor. The second task was to do lane following in the Udacity Simulator. The third was to drive towards a red marker and then make a U-turn around it.

3.2 SPURV Setup

This section concerns itself with the setup and usage of the SPURV, while explaining certain design choices. Any aspects related to the low-level hardware control and

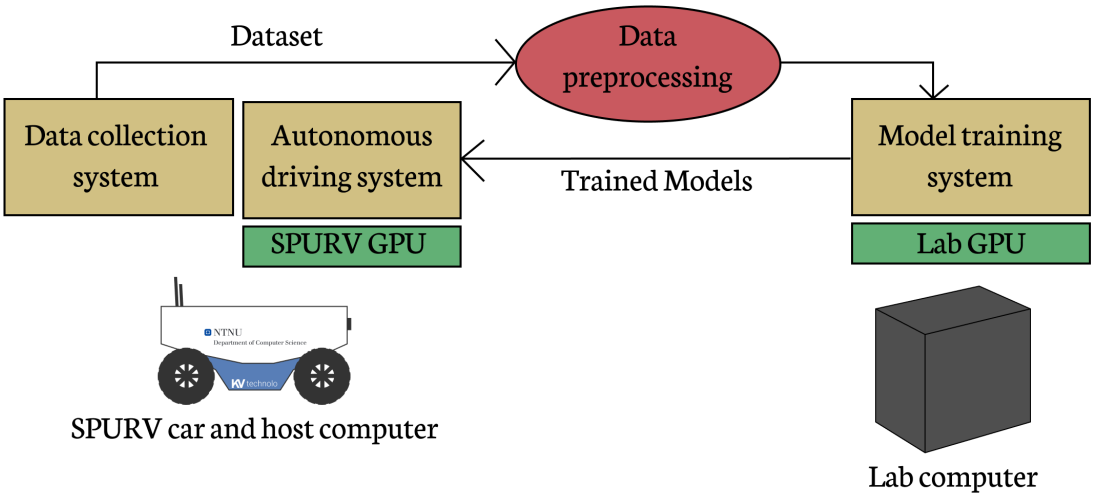


Figure 3.1: An overview of the components used in this research project.

steering of the car are not considered or heavily simplified, as they have not been a relevant part of this thesis. The full step-by-step usage guide for the SPURV is included in Appendix A.2.

3.2.1 Data Collection

The SPURV car was used for collecting training- and testing data for two of the tasks in this thesis. This section explains the details of how this process was carried out, focusing on the technical aspects related to the SPURV.

Manually Steering the SPURV

In the context of ROS-controlled robots, one distinguishes between the computer aboard the robot and another computer that has a wireless connection to the robot through which commands are sent from. This separate computer will from now on be referred to as the host laptop. ROS and all other relevant libraries and drivers were installed on the host computer.

The car was driven using the two joysticks on an Xbox controller that was connected to the host laptop. One joystick controlled the speed, while the other was used for turning. Figure 3.2 shows the complete ROS computation graph during data collection. Note that the *Spurv Joy Driver* node was implemented by KVS Technologies and delivered along with the SPURV. The max speed of the node was reduced to make it more suitable for indoor driving.

As seen in Figure 3.3, the human driver walked behind the SPURV while carrying the host laptop and steering using the Xbox controller. Due to technical difficulties with

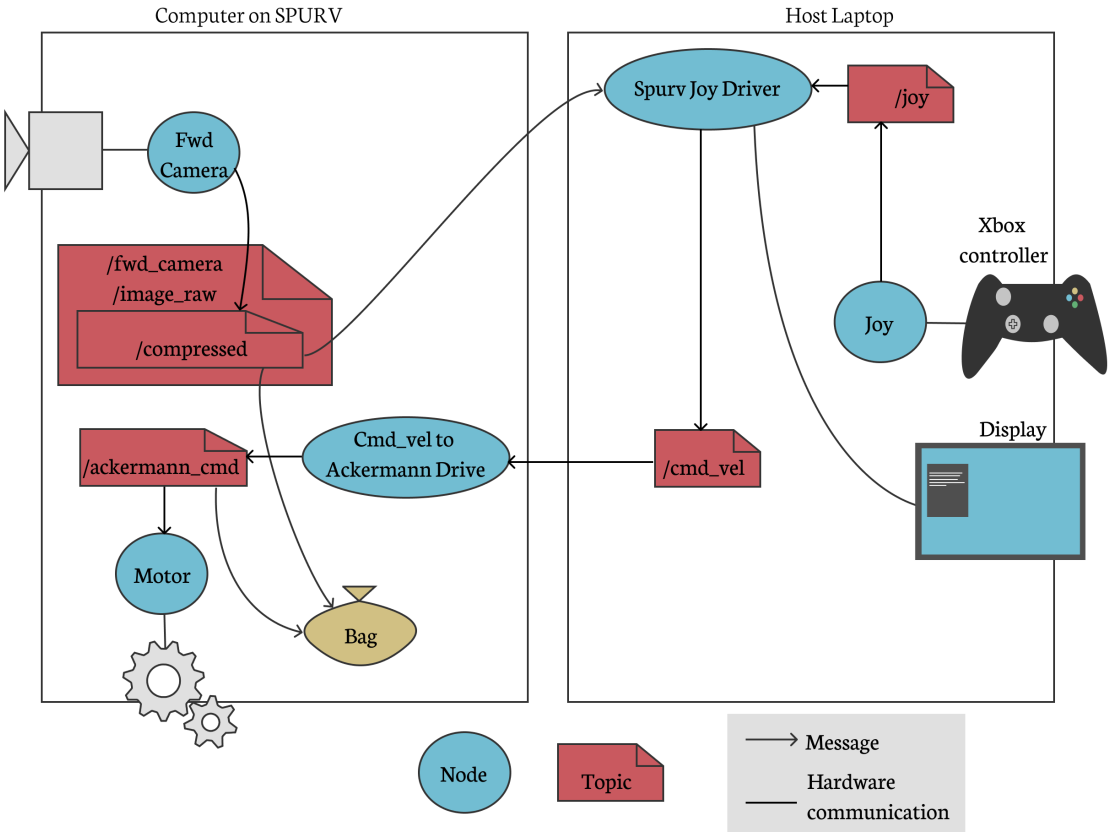


Figure 3.2: The ROS computation graph during data collection using the SPURV. The Xbox controller is used to steer the SPURV, while the display shows the images from the forward camera. Steering commands and images are saved in a ROS bag. Note that the details between `/ackermann_cmd` and the actuators are omitted.

the file transfer speed of images between the SPURV and the host laptop causing a massive time delay, remote steering where the human driver was only using camera images sent to the laptop to make decisions was deemed unsuitable.

Recording ROS Bags

The ROS computation graph during data collection can be seen in Figure 3.2. Training data was recorded in ROS bags that were saved on an SD card mounted in the SPURV, due to disc space issues. As saving the uncompressed raw images published to the `/fwd_camera` topic caused buffer overflows that could not be deterred by increasing the buffer size, the compressed images were recorded instead of the raw ones.

After the data collection was completed, the ROS bags were manually transferred to the host laptop. Here, they were inspected using the `rqt` tool that provides a simple user interface for playing bags and viewing the messages for the various topics in an intuitive way. It was discovered that the messages in the topics were not synchronised.

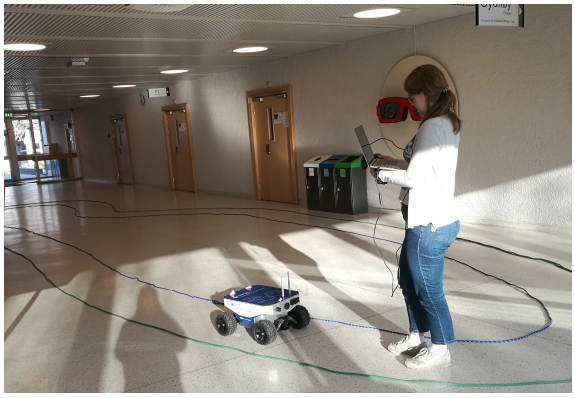


Figure 3.3: An image taken during the data collection process in the lane following task. One person walks behind the SPURV carrying the host laptop while simultaneously steering with an Xbox controller.

As neither images nor `/cmd_vel` messages had a timestamp from the moment they were created, they were assigned a timestamp when they were received in the ROS bag. Due to file Input/Output (I/O) delays, these timestamps were not coherent. This was solved by rewriting the forward- and backward camera nodes to include a timestamp in their published messages, while also switching to the `/ackermann_cmd` topic for saving steering commands.

The reasons for choosing the `/ackermann_cmd` topic for collecting steering commands were several: As seen in Figure 3.2, there were three possible steering-related topics. Choosing `/joy` would restrict the use case of the collected data and trained models to joystick-based controllers like the controllers for Xbox. As `/cmd_vel`, which is a standard steering command in ROS, has no timestamp included, there was no way of synchronising the various messages in this topic. Additionally, `/cmd_vel` uses `Twist` messages, which is specified by respectively a linear velocity vector and an angular velocity vector. This left `/ackermann_cmd`, where the steering angle is defined as the average of the angles of the two front wheels in radians (*ROS Wiki* n.d.). Note that the nature of Ackermann steering gives left turns positive values, while right turns are negative, as explained in Figure 3.4.

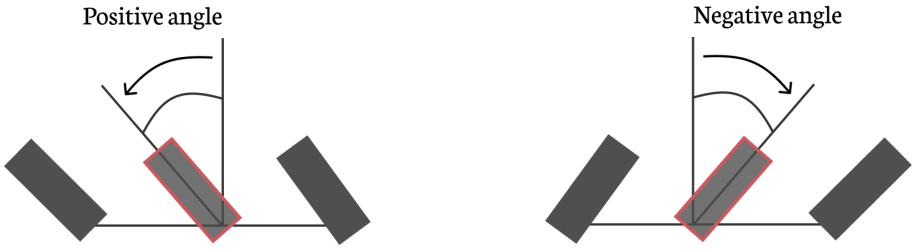


Figure 3.4: An illustration of what the Ackermann steering angles in ROS represent. The imaginary wheel with the red border represents the Ackermann steering value. The red wheel is rotated with the average angle of the front left and right wheel. When the car is turning left, the steering angle is positive, and its value is negative if the car is turning right.

As ROS bags must be played in order to retrieve their data, they are not suitable for training deep learning systems. Thus, the data was transformed to an appropriate structure. A script that listens to messages from a recorded ROS bag being played was provided by Revolve NTNU, and slightly modified to fit the chosen file structure for this project. Upon receiving a message containing an image, the script saves the image file in a separate folder, reads the latest steering command it has received and stores it along with the path to the image as a line in a CSV file.

3.2.2 Autonomous Driving

To enable real-life testing of the trained models, a new ROS node, *Autonomous Drive* was developed. This node had to run on the SPURV to access the onboard GPU. CUDA was installed on the SPURV through NVIDIA JetPack, followed by the installation of TensorFlow and Keras. To address safety concerns, the Xbox controller was set up to start and stop the autonomous driving manually, making it work as a security switch. The ROS computation graph during autonomous driving is illustrated in Figure 3.5. The node listens to the `/fwd_camera/image_raw` topic. When autonomous driving is activated, the received images are cropped, resized and normalised, as described in detail in Section 3.3, before being sent through the network. The predicted steering angles are then published to the `/ackermann_cmd` topic along with a constant speed of 0.4 m/s. The full implementation of the node is included in Appendix B.3.

For the marker turning task, the Autonomous Drive node was modified to make the *Spatial History Autonomous Drive* node. The Spatial History Autonomous Drive node stores past images to be able to feed a spatial history of three images into the network at each prediction. The enlarged network size caused a memory issue, which was solved by setting an upper bound on the fraction of memory that is made available to each TensorFlow process.

3.3 Training Setup

This section covers the various aspects of training the neural networks. It covers the hardware the networks were trained on, the metrics they were evaluated with, various regularisation and optimisation techniques used. The data processing steps are also described here.

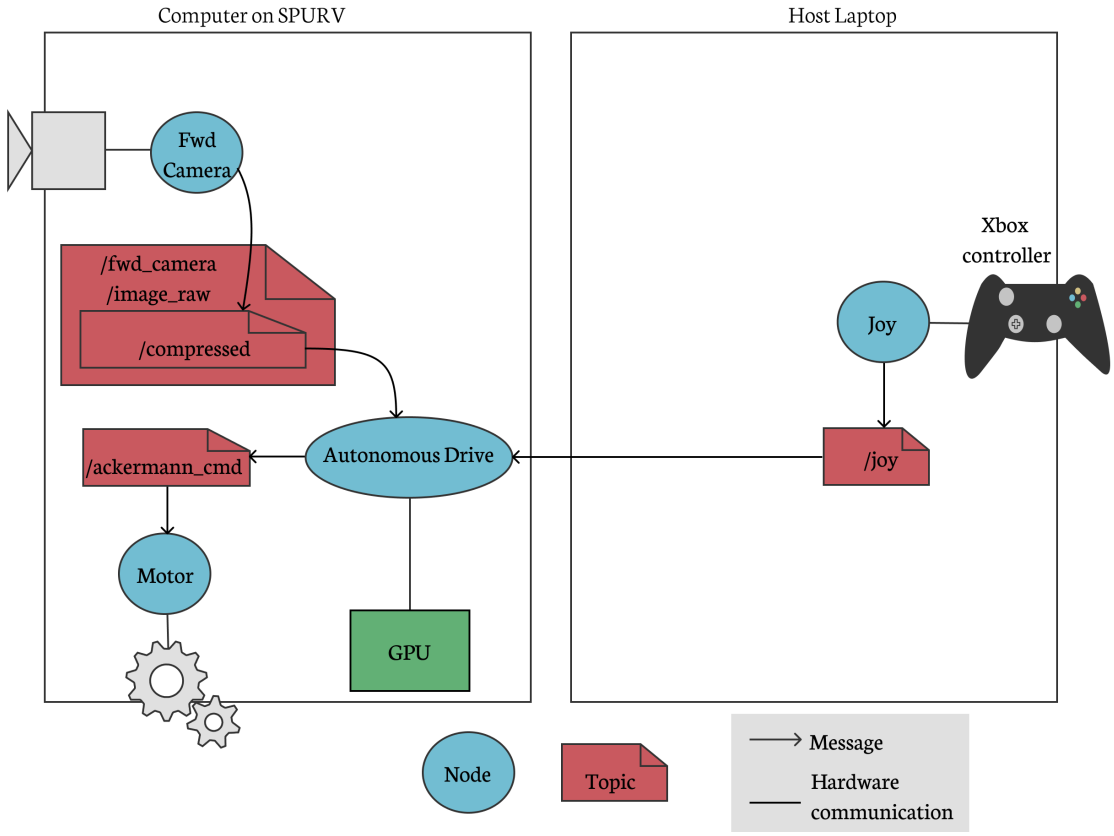


Figure 3.5: The ROS computation graph during autonomous driving using the SPURV. The Xbox controller is used only for turning on and off autonomous mode, and does not output steering commands. Note that the details between `/ackermann_cmd` and the actuators are omitted.

3.3.1 Training Environment

All training was done on an NVIDIA Titan X GPU. TensorFlow, Keras and other relevant libraries were installed within a Docker container, which was run using `nvidia-docker` in order to get access to the GPU. See Appendix B.1 for the complete Docker setup.

Initially, training was slow due to I/O issues with reading the image data to the GPU's internal memory. Experimenting with the batch size while also reducing the size of the original image files pushed the training time to an acceptable level. All training was done with a batch size of 64, and required between one and two hours to complete when trained between 10 to 20 epochs.

Mini-batches

Random mini-batch gradient descent was used as only a small subset of the samples fit in memory. This was implemented using the `fit_generator` method from the Keras API. Each batch was generated by a unique, random selection over the entire training set. As mentioned, the mini-batch size was 64. The number of steps per epoch, i.e., the number of mini-batches trained on during one epoch was set to

$$\text{steps per epoch} = \frac{\text{number of training samples}}{\text{mini batch size}}, \tag{3.1}$$

which makes the epoch-length equal to the length of the training data.

Metrics

This section summarises the various metrics used to quantitatively assess the models during training. MSE, as defined in Equation 2.2, was used as the training loss.

In addition, the mean absolute error (MAE) was monitored during training, as it provided a more intuitive interpretation of the results. MAE is defined as

$$M(t, y)_{MAE} = \frac{1}{n}\sum_{i=1}^{n}(|y_i - t_i|), \tag{3.2}$$

where n is the total number of elements in y and t.

Early Stopping

Early stopping was used during training with a patience of seven, meaning that if the validation loss did not decrease for seven epochs, training would be stopped. Models checkpoints were saved every three epochs, making it possible to revert to the model at an earlier stage of training in the case of overfitting.

Dropout

Dropout was applied to all hidden layers in the models. Through experimentation, a dropout rate of 0.35 was chosen.

Optimisers and Learning Rate

Experiments using the three different optimisers SGD, RMSProp and Adam and various initial learning rates were conducted to compare their performance on the

problem. Based on a quantitative comparison of the MSE and MAE curves during both training and validation, and a qualitative measure on comparing the VBP masks to detect severe overfitting, Adam with an initial learning rate of 0.0003 was chosen to train all models.

3.3.2 Data Processing

The following sections give an overview of the data processing pipeline, including filtering out corrupt data, splitting data into training, validation and test sets and the preparation of the images.

Removing data points with zero speed

The data recordings contained some periods of time where the car was stopped, due to the starting and stopping of the recording or external factors interrupting the driving. Every data entry with zero speed was filtered out and discarded, as they would add noise to the dataset.

Splitting Data Into Training, Validation and Test Sets

The splitting of the gathered data into training and validation set was first done by randomly selecting individual frames from the whole training data set. This was found to be a bad choice because the frame rate of 15 fps used during data gathering resulted in almost identical data points being distributed between the training and validation set. The training loss and validation error were unrealistically low since the validation set was unable to reveal overfitting. The validation set was, therefore, a set of data collected from a completely new and unseen track.

Image Preparation

The upper parts of the images show only the roof and walls and were discarded, to reduce computational complexity without losing any relevant information. By inspection, it was found that 35% could safely be removed from the top of the images, shown in Figure 3.6.

Each image was fed into the network as a three-dimensional (blue, green, red) (BGR) image array, of size equal to the input layer size, which was 66 pixels high and 200 pixels wide. The size of the images was reduced to match the size of the input layer.

Figure 3.6: Everything above the red line, the top 35%, was considered irrelevant and removed from the images.

Augmentation methods were applied to the images in the training set during training. These are described in further detail in Section 3.3.3.

Finally, the image arrays were normalised before being sent as input to the network. Each BGR channel value was reduced to the $[-0.5, 0.5]$ interval, by the following equation:

$$\text{image array} = \frac{\text{image array}}{255} - 0.5. \tag{3.3}$$

3.3.3 Augmentation

This section describes the different augmentation techniques used on the training set. Which of the augmentation techniques that got applied to each image was chosen randomly for each mini-batch, and all of the techniques could be combined in the same image, producing a large number of new samples. The probabilities of each augmentation method being chosen are henceforth referred to as the $P(\text{method})$. The selection probabilities were found by experimentation, and are specified in the corresponding sections. The intensity of each augmentation technique, for instance, how many pixels an image was shifted when applying horizontal shifts, was also found by experimentation. The intensity was for some methods subjected to randomness, in which case an *intensity range* is specified.

Flipping

The images were flipped and the corresponding steering angles multiplied by -1, as illustrated in Figure 3.7. This was done to increase the number of training samples and ensure that the data was not unbalanced toward turning in either direction. The probability was:

$$P(\text{flipping}): \quad 0.5$$

(a) -0.27 (right) steering angle **(b)** 0.27 (left) steering angle

Figure 3.7: Image (b) is a flipped version of image (a). The steering angle has been multiplied by -1.

Hue, Saturation and Value adjustments

The brightness level of the images in the training data varied greatly, as data were gathered at different locations and at different times of the day. Brightness levels are also affected by the position of the car relative to the windows and other light sources. The saturation and colour values were also affected by the light conditions. Random adjustments of the brightness, saturation and colour levels of training images were done to make the model more robust to these changes. The image was converted to the (hue, saturation, value) (HSV) colour model, and each of the three values was multiplied independently by a random real number in the ranges specified below. Examples of different HSV changes are shown in Figure 3.8. The probability and intensity ranges were:

$$\begin{aligned} P(\text{HSV adjustments}): &\quad 1 \\ \text{Brightness(value) intensity range}: &\quad [0.5, 1.5] \\ \text{Saturation intensity range}: &\quad [0.5, 1.5] \\ \text{Colour(hue) intensity range}: &\quad [0.9, 1.1] \end{aligned}$$

Horizontal shifts

All of the data gathered is a result of behaviour that seeks to be ideal: driving in the middle of the road. Since the model was going to interact with its environment, it needed to be trained for unideal situations as well, like being away from the middle of the road. This way, the model is taught how to return to correct positioning

(a) Original image

(b) 1.5 × value

(c) 0.5 × value

(d) 1.1 × hue

(e) 1.5 × saturation

(f) 0.78×value, 1.05×hue, 1.45×saturation

Figure 3.8: Different HSV adjustments of image (a).

once diverged from the middle. Images were therefore shifted vertically and a corresponding offset was added to the steering angle, as shown in Figure 3.9. A shift of the image to the right, simulates the car being in the left part of the lane and the steering angle needs to turn harder to the right. More specifically, a shift to the right caused an angle update according to the equation below (and vice versa for left shifts). Note that since the Udacity steering angles had magnitudes opposite to the SPURV's steering angles, the equation was inverted for the Udacity application.

$$\text{Shifted angle} = \text{angle} - 0.0065 \times \text{number of pixels shifted to the right.} \tag{3.4}$$

The probability and intensity range was:

$$\begin{aligned} P(\text{Horizontal shift}): &\quad 0.7 \\ \text{Intensity range}: &\quad [-40, 40] \text{ (pixels)} \end{aligned}$$

Figure 3.9: In the image to the right, the example image (left) has been translated to the right, and the missing pixels are replaced by black pixels. The steering angle is also shifted to the right by a number proportionate to the pixel size of the shift.

Random erasing

Random erasing was used to battle overfitting and to help the model handle images with regions that have been destroyed by light and reflections of light, as shown in Figure 3.10. A randomly sized and shaped rectangle was erased from the image at a random location by colouring all pixels within the rectangle white. White erasing was chosen to further mimic light reflections. The probability and intensity range was

$$\begin{aligned} P(\text{P(Random erasing)}: &\quad 0.2 \\ \text{Size intensity range}: &\quad [2, 40] \text{ (percentage of image area)} \end{aligned}$$

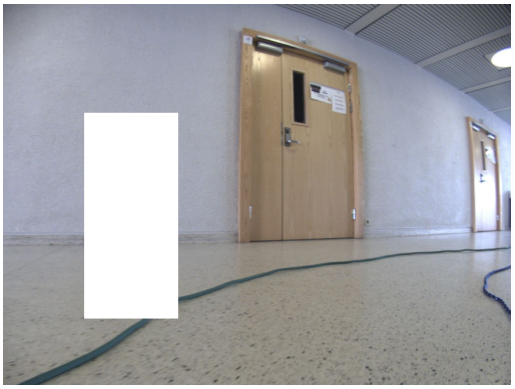



Figure 3.10: To the left an example of an image that suffers from occlusion from light reflection. To the right is an example of an image augmented by erasing a random square.

3.3.4 Evaluation

In addition to the metrics introduced in Section 3.3.1, the trained models were evaluated in the context of whiteness and two qualitative evaluation metrics. These are briefly summarised in this section.

Whiteness

Whiteness, as introduced in, Section 2.2.3 was calculated to quantify the stability of the trained models. Equation 2.4 was modified to

$$W = \frac{1}{n}\sum_{i=1}^{n-1}(y_{i+1} - y_i)^2, \tag{3.5}$$

where y_i is the predicted steering angle at step i, and n is the number of predictions in the validation set.

Qualitative Evaluation

In addition to the quantitative metrics during training and evaluation, some qualitative evaluation techniques such as visual inspection of the visualisations, further described in Section 3.4, were included. The VBP masks generated for each image in the validation set were examined. The person examining the masks registered whether the model seemed to focus on relevant features in the images, or if irrelevant features appeared to have a large impact on the predictions. Also, the car path visualisations were investigated in order to see if the models seemed to be able to complete their respective tasks.

Real-life Evaluation

A final test in real-life conditions was conducted on the models with the most promising results in the prior evaluation. A varied track relevant to the task was set up just like during data collection. The SPURV would drive in the track under careful observation. The camera input and steering commands made by the model were recorded in order to enable inspection later on. VBP heat maps were created from these recorded images after the real-life test.

3.4 Visualisation

This section describes the various visualisations techniques that were utilised throughout this thesis as an attempt to reason about the trained models' performance. Section 3.4.1 describes the method used to visualise the future path of the car in a set of images, while Section 3.4.2 explains the implementation of VBP.

3.4.1 Car Path Visualisation

When running predictions on a test set, it can be useful to visualise portions of the future path of a car in an image. This can aid validation of the model's performance by giving a more nuanced picture of the car's behaviour. An accurate version of such a path can be generated by utilising trigonometry and vector algebra. A trigonometrical illustration of the problem can be seen in Figure 3.11. Let's assume that an image I_0 captured by the front camera has width w_{IRL} and height h_{IRL} in real life. Here, d_0 is the distance from the front of the car to the first point on the floor that is visible in the front camera's field of view in image I_0. Assuming that the distances h and d_0 can be measured, the tangents of the angles α_0, α_1, α_2 and so on can be calculated as follows:

$$\tan(\alpha_0) = \frac{h}{d_0}, \quad \tan(\alpha_1) = \frac{h}{d_0 + d_1} = \frac{h_1}{d_1}, \quad \tan(\alpha_2) = \frac{h}{d_0 + d_1 + d_2} = \frac{h_2}{d_2}.$$

Here, d_i for $i \in \{1, 2, 3, ...\}$ is calculated by using the car's speed and the frame rate of the images. Generally, the tangent of a given angle α_i can be calculated as

$$\tan(\alpha_i) = \frac{h}{\sum_{j=0}^{i} d_j},$$

and the heights h_i, being on the opposite side of α_i are simply

$$h_i = \tan(\alpha_i) \times d_i.$$

From this, it is clear that the heights h_i, where $i \in \{1, 2, 3, ...\}$, can be calculated by

$$h_i = \frac{h \times d_i}{\sum_{j=0}^{i} d_j}. \tag{3.6}$$

In order to transfer the heights in Equation 3.6 to the perspective of image I_0, the vectors $\mathbf{r}_i$ are introduced for $i \in \{1, 2, 3, ...\}$. The vector $\mathbf{r}_i$ maps to the scalar h_i by

$$h_i = |\mathbf{r}_i|. \tag{3.7}$$

For every vector $\mathbf{r}_i$, a corresponding reference vector $\mathbf{r}_i^*$ is introduced, where $|\mathbf{r}_i^*| = |\mathbf{r}_i|$ and $\mathbf{r}_i^*$ is parallel to $\mathbf{r}_{i-1}$. The reference vector represents the direction of the car when it has completed turn θ_{i-1}. The relationship between the path vector and reference vector is illustrated in Figure 3.12. The first reference vector, $\mathbf{r}_1^*$, is defined as

$$\mathbf{r}_1^* = \begin{bmatrix} 0 \\ |h_1| \end{bmatrix},$$

making it perpendicular to the x-axis. The following reference vectors $\mathbf{r}_i^*$ for $i \in \{2, 3, ...\}$ are calculated by extending the vector $\mathbf{r}_{i-1}$ and then removing the length of $\mathbf{r}_{i-1}$:

$$\mathbf{r}_i^* = \mathbf{r}_{i-1} \times \frac{h_i + h_{i-1}}{h_{i-1}} - \mathbf{r}_{i-1}.$$

Each reference vector $\mathbf{r}_i^*$ is rotated by the angle θ_i in order to obtain $\mathbf{r}_i$

$$\mathbf{r}_i = R(\theta_i) \times \mathbf{r}_i^*, \tag{3.8}$$

where $R(\theta)$ is defined as the rotation matrix

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}. \tag{3.9}$$

Finally, as shown in Figure 3.12, the vectors p_i are calculated in order to obtain the coordinates of each point to plot in the path visualisation:

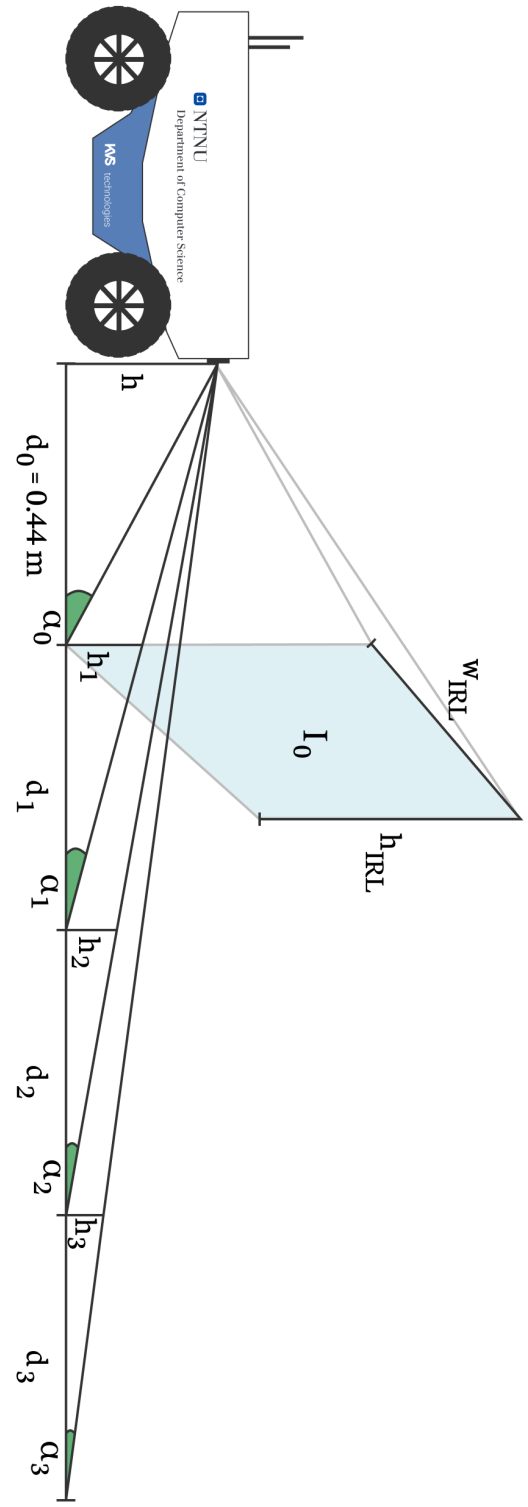


Figure 3.11: Trigonometrical illustration of how future steps the SPURV car makes are perceived in image I_0.

$$\mathbf{p}_i = \mathbf{p}_{i-1} + \mathbf{r}_i. \tag{3.10}$$

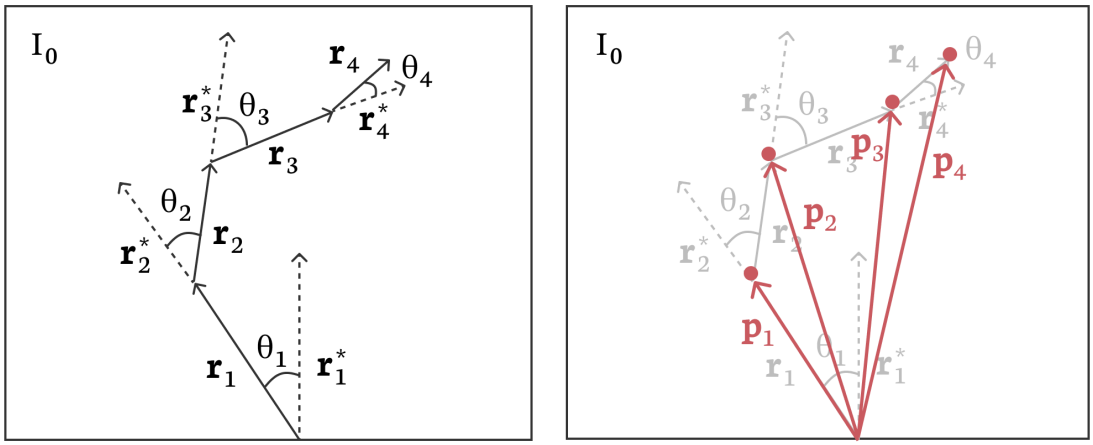


Figure 3.12: Left: An illustration of I_0 with future car paths represented as vectors $\mathbf{r}_i$. Right: An illustration of how the sum $\mathbf{p}_i$ of each vector $\mathbf{r}_i$ directly maps to the coordinates of points in the path visualisation.

The coordinates $\mathbf{p}_i$ are in metres. The last thing that remains is scaling them to the actual pixel size of the image they are to be plotted on. Defining

$$\mathbf{s}_{IRL} = \begin{bmatrix} w_{IRL} \\ h_{IRL} \end{bmatrix}, \quad \mathbf{s}_{px} = \begin{bmatrix} x_{px} \\ y_{px} \end{bmatrix},$$

where x_{px} and y_{px} are the plotted image's size in pixels, the final cartesian coordinates in pixels $\mathbf{p}_i^{IRL}$ for $i \in \{1, 2, 3, ...\}$ can be calculated by

$$\mathbf{p}_i^{IRL} = \frac{\mathbf{p}_i \times \mathbf{s}_{px}}{\mathbf{s}_{IRL}}.$$

Practical considerations

While this visualisation method works in theory, there are some aspects that give it some inaccuracy in practice. The first aspect is the angles themselves. Each image is captured at 15 frames per second. Although the corresponding steering command states that the car has turned with an angle θ, it has not, in fact, turned the full θ radians in 1/15th of a second. Measurements show that the SPURV needs 1.4 seconds to complete a full turn where it is aligned with the steering angle in the command. This requires the introduction of an angle scaling factor $f_\theta = \frac{1.4}{15}$

Equation 3.8 is thus redefined to:

$$\mathbf{r}_i = R(\theta_i * f_\theta) \times \mathbf{r}_i^*, \tag{3.11}$$

which scales the predicted angle to the actual angle the SPURV is able to turn in 1/15th of a second.

Another factor causing inaccuracy is the fact that the wide angle of the front camera on the SPURV causes some distortion in the image. While the camera captures the same width throughout the whole image, the height is distorted. This can be seen in Figure 3.13, where the top and bottom horizontal lines appear curved in the edges of the image. This causes the dots to not directly map to the Cartesian coordinate system as earlier assumed, adding some more inaccuracy.

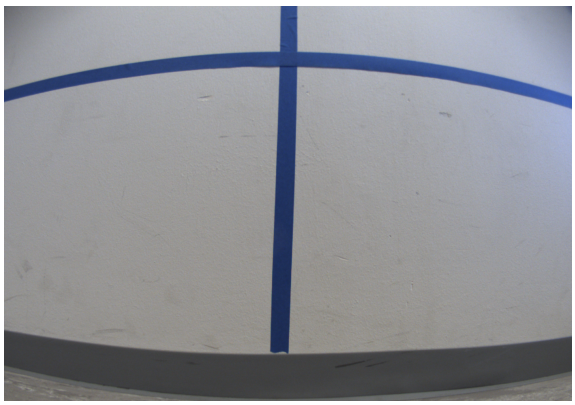


Figure 3.13: An example of the fish eye-like distortion in the forward camera on the SPURV. In reality, all lines in this scene are straight.

Although these inaccuracies are present, the method gives insight into the path of the car in the test set. It can also help compare the predictions of a model to the ground truth over a sequence of frames. Figure 3.14 is an example of such visualisation. The complete implementation in the form of a Jupyter notebook can be found in Appendix B.4.

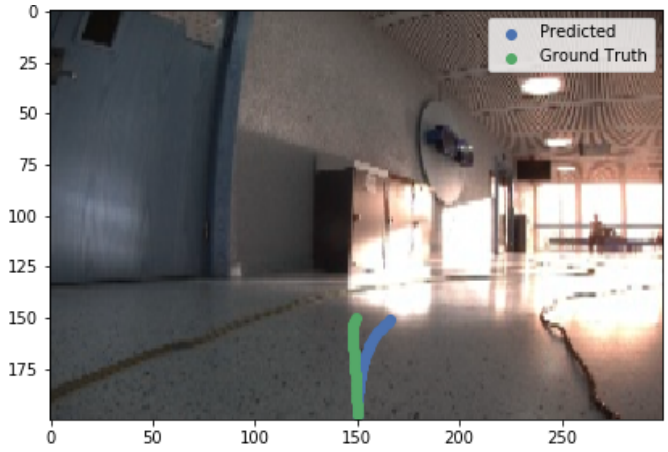


Figure 3.14: Arbitrary output from the path visualisation implementation. The two paths illustrate the approximate path of the car two seconds ahead in time, for both the ground truth and the predictions.

3.4.2 CNN Visualisation

At the time of writing this thesis, there were no suitable implementations of VBP, as described in Section 2.1.3, available to the public. The Lua[1] implementation by Bojarski, Choromanska, et al. (2016) was considered, but this solution was implemented in Torch[2]. There was no intuitive way of translating the models in this thesis, which were built in Keras, to Torch models. A few Keras implementations were available online, but these were hard-coded to only support specific CNN architectures like VGG-16. As there was a wish to test various architectures in this thesis, a general version of the VisualBackProp visualisation method was implemented in Python. Figure 3.15 shows a result of the implementation. The complete implementation can be found in Appendix B.2.

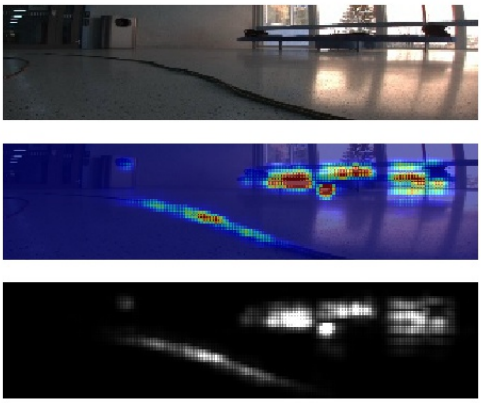


Figure 3.15: An arbitrary output of the VBP implementation. The visualisation shows that the network focuses on wanted features in the lane, but also reveals undesired focus on the benches in the back of the image.

3.5 SPURV Lane Following

This section describes the methods used specifically for making the SPURV autonomously follow an indoor lane, simulated by laying two ropes on the floor.

3.5.1 The Dataset

To gather data, two ropes were placed in parallel on the floor to form tracks. The distance between the ropes ranged from 70 to 110 cm, and the length of the tracks

[1] `https://www.lua.org/`
[2] `http://torch.ch/`

ranged from 20 to 30 meters. Care was taken to create tracks with a variety of turn layouts, while maintaining a relatively equal distance between lanes. Eight different tracks were made in three different indoor locations, shown in Figure 3.16. Each track was driven three to five times in each direction. The ropes curled and bent easily, making some irregularities along the lanes, as can be seen in 3.16(c). The car lost sight of the inner rope during steep turns, i.e. turns where the steering angle was kept at 0.3 radians or larger for some period of time.

As described in Section 3.2.1, manual steering was, due to technical issues, done from the point of view of the driver walking behind the SPURV and not by looking at the live video from the SPURV's point of view. When driving, it was sought to keep the car in the middle of the lane at all times. Turning smoothly, attempting to use steering angle values proportionate to the steepness of the turns, was another focus of attention to capture behaviour that was as consistent as possible. This was complicated by a somewhat sensitive joystick. The joystick further seemed to have a slight bend to the right when in a neutral position, which made driving straight distances particularly difficult.

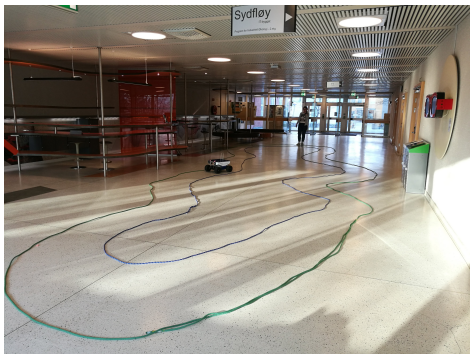
After processing, the final dataset consisted of 85986 data points or 96 minutes of recordings. The data was split into a training set of size 64294 (75% of total dataset), a validation set of 7129 (8%) and a test set of 14563 (17%) data points. A more detailed overview is available in Table 3.1.

Table 3.1: The different tracks and locations of the SPURV lane following dataset. The locations are pictured in Figure 3.16.

	Track	Location	Number of data points	≈ minutes
Training	Track 1	Location A	9606	10
	Track 2	Location A	6837	9
	Track 3	Location B	9261	10
	Track 5	Location B	16 061	18
	Track 6	Location B	12 133	13
	Track 8	Location C	10 396	12
Validation	Track 4	Location B	7129	8
Testing	Track 7	Location B	14563	16

Upsampling large angles

The collected data was heavily biased towards small turning angles. This caused the trained models to be the same. Additionally, the models were punished during training for large angle deviation, which made them favour small angle predictions. To avoid ending up with models only predicting small angles, the data points with absolute steering angle larger than 0.5 were upsampled by a factor of five in the



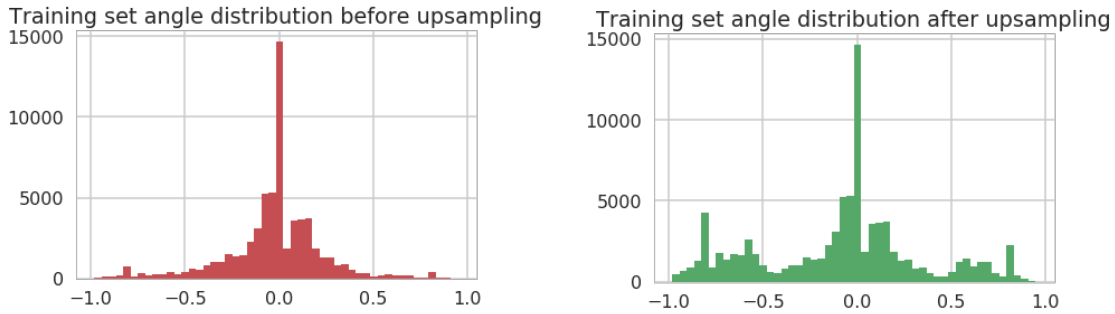
(a) Location B

(b) Location A

(c) Location C

Figure 3.16: The three different indoor locations where data was collected for the SPURV lane following task. In the picture to the right, one can see that the left rope is bent and curled at some points, making the lane irregular.

training set. The slight improvement in the distribution of angles in the training data before and after upsampling can be viewed in Figure 3.17.



(a) Before upsampling **(b)** After upsampling

Figure 3.17: Histograms showing the distribution of angles of the ground truth in the validation set for the SPURV lane following task before and after the upsampling of angles larger than 0.5 radians by a factor of 5.

3.5.2 Architecture

Table 3.2 shows the various layers in the *SpurvPilot* model, which is based on PilotNet by Bojarski, Del Testa, et al. (2016). This architecture was used for the lane following tasks.

Table 3.2: Overview of the various layers in the *SpurvPilot* model, based on Bojarski, Del Testa, et al. (2016).

Layer type	Kernels@Kernel size	Stride size	Output size	Activation
Input	-	-	3×200×60	-
Conv2D	24@5×5	2×2	31×98×24	ReLU
Conv2D	36@5×5	2×2	14×47×36	ReLU
Conv2D	48@5×5	2×2	5×22×48	ReLU
Conv2D	64@3×3	1×1	3×20×64	ReLU
Conv2D	64@3×3	1×1	2×18×64	ReLU
Flatten	-	-	1152	-
Fully Connected	-	-	100	ReLU
Fully Connected	-	-	50	ReLU
Fully Connected	-	-	10	ReLU
Fully Connected	-	-	1	Linear

3.6 Lane Following in the Udacity Simulator

This section is concerned with the research aspects related to the lane following task in the Udacity Simulator.

3.6.1 The Dataset

Data was gathered on the *Lake Track*, which was the simplest of the two available tracks, in the Udacity Self-Driving Car Simulator. This resulted in 20946 (80% of total dataset) data points in the training set and 5144 (20%) data points in the validation set. By default, the simulator collects data at 10 fps. An effort was made to stay in the middle of the lane at all times during data collection. A dataset released by Udacity, containing 8037 elements, was chosen as test set. The angle distribution of the test set is shown in Figure 3.18

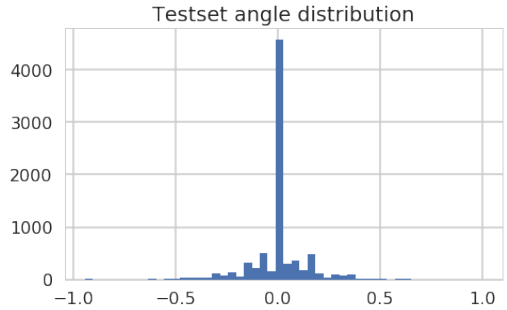


Figure 3.18: Angle distribution of the Udacity Simulator test set.

Side Camera Images

In addition to the forward facing images, the Udacity simulator automatically saves data from two imaginary side-facing cameras for each data point, shown in Figure 3.19. These can be used to simulate the car being rotated slightly to the right, in a similar manner as the horizontal shifts described in Section 3.3.3. Through experimentation, an offset of 0.25 was added to the steering angle when side cameras were used. During training, either of the left, right or centre images had an equal chance of being included in each batch.

3.6.2 Architechture

The *SpurvPilot* architecture used for the SPURV lane following task, specified in Table 3.2, was also used for lane following inside the Udacity Simulator. The

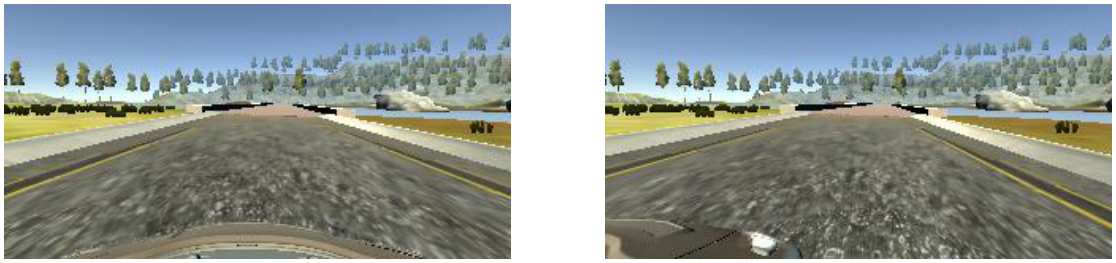


Figure 3.19: The left image shows the view from the forward facing camera, compared to the view from the right side-camera, shown in the image to the right.

weights of the trained SPURV lane following model were used to initialise the Udacity Simulator model, in order to reduce the risk of overfitting and investigate if the desired properties of the *SpurvPilot* model could be transferred to the Udacity Simulator case. This model is denoted as $SpurvPilot_{Udacity}$ in this thesis.

3.6.3 Testing

As testing in the Udacity Simulator is more easily available than real-life testing on the SPURV, no attempt at adapting the car path visualisation explained in Section 3.4.1, to the Udacity case, was made. Instead, all testing was done in the Udacity simulator. The models were tested on the Lake Track.

3.7 Marker Turning Task

This section covers the methods used specifically for making the SPURV make a U-turn based on visual cues only. The visual cue consisted of a red marker placed randomly on the floor. The task was to drive towards the marker and make a U-turn around it. During the U-turn, the SPURV would not be able to see the marker it was currently turning around. Upon seeing a new marker, the SPURV was to drive towards it. The idea was to use spatial history, as introduced in Section 2.2.2, to make the model recall if there were any markers in its surroundings. If no markers were present in the model's spatial history, the car was to drive straight forward. As the SPURV was set to drive at constant speed in this task, the spatial history distance could be set to a constant number of frames. See Figure 3.22 for an explanation of the task in the context of spatial history with three images.

3.7.1 The Dataset

Two people were necessary for the data collection. The driver walked behind the SPURV while steering it, and conformed to the following rules:

- Drive at a constant speed of 0.5 m/s.
- If a marker is in the SPURV's field of view, drive directly towards it.
- When the SPURV is approximately 0.5 meters away from the marker, turn max to the right until the front of the SPURV is aligned with the marker, then turn max to the left until next marker is in the SPURV's field of view.
- If there are no markers anywhere, drive straight.

Figure 3.20: The data collection setup in *Location D*.

The two markers used were made out of an A1-sized sheet of red cardboard rolled into a cylinder shape. The other person was responsible for placing the markers in the room. As the SPURV was driving away from a marker it had just made a U-turn around, the given marker would be moved to a different location. This would thus happen outside of the SPURV's view. This way, the dataset would include a variety of marker locations while simultaneously being a set of long, continuous sequences of images. See Figure 3.21 for an illustration of the data gathering process. Figure 3.20 shows the gathering process in *Location D*.

In order to collect data for the case when there were no markers around the car, i.e. not in its spatial history, data of driving straight in rooms without the markers was collected. Here, the driver would steer the car in straight lines, backing up and turning randomly around when reaching a wall. Any driving beside the portions where the car was driving straight ahead was filtered out from the training data.

The dataset statistics can be found in Table 3.3. In total, the final training set consisted of 28847 (81% of total dataset) data points, equivalent to 32 minutes. The validation set was 6799 (19%) data points large, which corresponds to 8 minutes. No test set was collected.

Data Augmentation

Due to the complicated nature of the task, a majority of the augmentation techniques described in Section 3.3.3 were deemed unfit for training the *SpatialSpurvPilot*

Table 3.3: The different tracks and locations of the SPURV marker turning task dataset.

	Take	Location	Number of data points	≈ minutes
Training	Take 1	Location B	6722	7
	Take 2	Location B	5076	6
	Take 3	Location D	2144	2
	Take 4	Location D	7333	8
	Take 5	Location D	4128	5
	No markers	Location D	3444	4
Validation	Take 6	Location B	6799	8

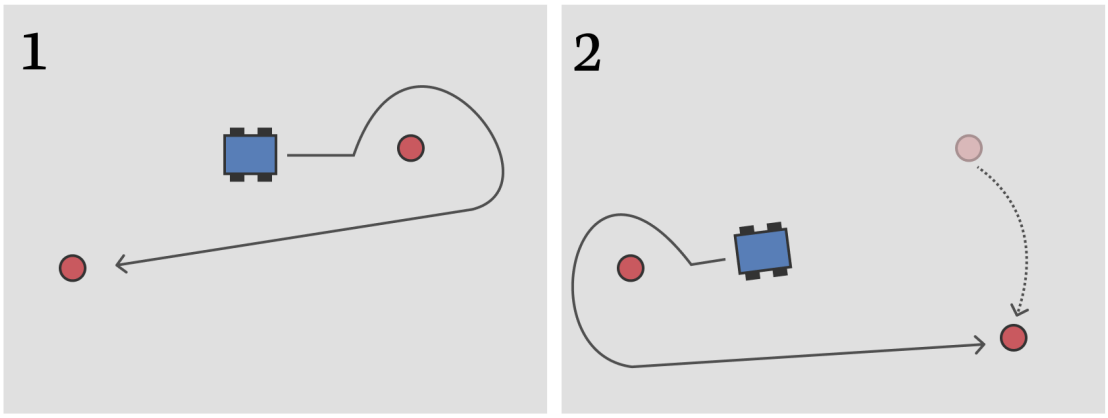


Figure 3.21: Illustration of two steps of the data gathering process for the marker turning task.

and *SpatialSpurvPilotExtended* models. Hence, only HSV adjustment was used for data augmentation. The reason for not using random erasing, was the risk of rendering a training sample invalid by accidentally covering an entire marker in the spatial history.

3.7.2 Spatial History

The input to the network consisted of three individual images appended to each other from left to right, where the leftmost image was the most recent one. The consecutive images were D_f image frames apart from each other in time, as in Figure 3.22. Three images were chosen, as it could not be guaranteed that a marker in the car's surroundings would be present in any of the images within the spatial history if only two images were submitted as input.

To make an informed choice on the D_f parameter, the time of the maximal possible turn the SPURV would have to make was measured. It was decided that the maximal turn was $2\pi + \frac{\pi}{4} = \frac{9\pi}{4}$ radians large, as illustrated in Figure 3.23. This limit was set before data collection was conducted, and the person moving the markers was thorough to not violate it. The time from the last moment the marker was visible in the front camera of the SPURV, until the maximal turn was completed and the next marker was visible, was measured to be approximately 11 seconds. Hence, $D_f = \frac{11s * 15\text{frames}/s}{2} \approx 80$ frames, with a data collection frame rate of 15 fps.

Architecture

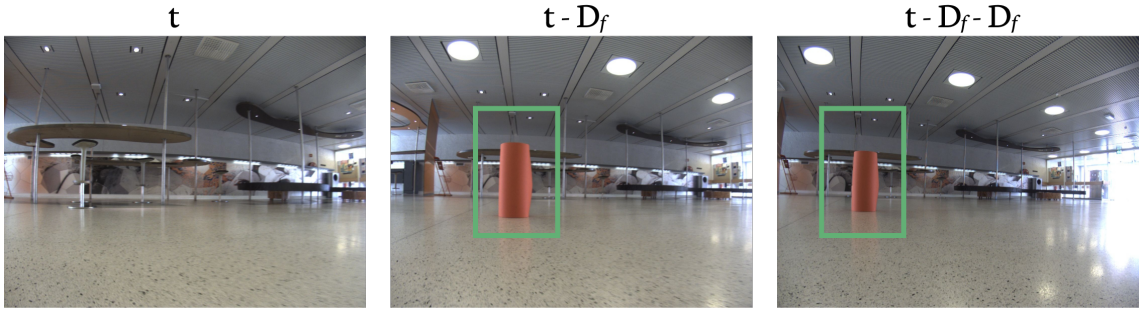
For the marker turning tasks, the *SpatialSpurvPilot* and *SpatialSpurvPilotExtended* architectures, shown in Table 3.4 and Table 3.5 respectively, were used. The main difference from *SpurvPilot* is the size of the input images, and that they both use the Tanh activation function. Additionally, *SpatialSpurvPilotExtended* has an additional fully connected layer, inspired by Hubschneider et al. (2017).

3.8 Summary

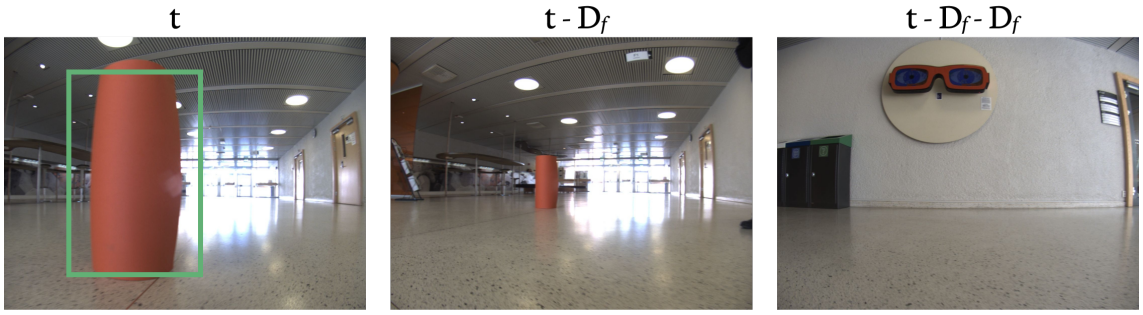
This chapter has described how data has been collected in order to use steering angle predictors to solve three respective tasks:

- An indoor lane following task on the SPURV robot, where tracks have been created by putting a pair of ropes on the floor to simulate outdoor lanes
- A lane following task in the Udacity simulator

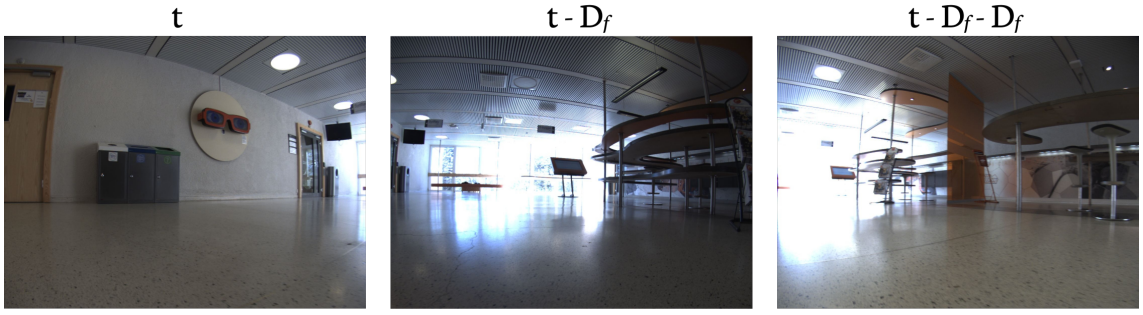
(a) Spatial history case where the SPURV should drive towards the marker



(b) Spatial history case where the SPURV should make a left turn



(c) Spatial history case where the SPURV should make a right turn



(d) Spatial history case where the SPURV should drive straight

Figure 3.22: Summary of the four spatial history cases where each of the three images in history are $D_f = 80$ frames apart. t is the most recent image in the history. The green rectangle denotes what the network should focus on when making the given choice. Whenever there is a marker in image t, that marker should be the one the network is focusing on, regardless of markers present in previous images.

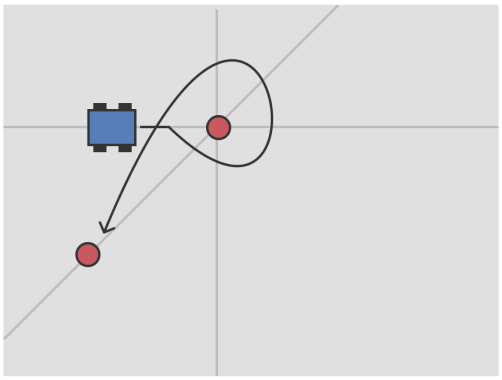


Figure 3.23: The maximal turn of $2\pi + \frac{\pi}{4} = \frac{9\pi}{4}$ radians the SPURV would have to make.

Table 3.4: Overview of the various layers in the SpatialSpurvPilot model, based on Bojarski, Del Testa, et al. (2016) and Hubschneider et al. (2017).

Layer type	Kernels@Kernel size	Stride size	Output size	Activation
Input	-	-	3×600×60	-
Conv2D	24@5×5	2×2	31×298×24	ReLU
Conv2D	36@5×5	2×2	14×147×36	ReLU
Conv2D	48@5×5	2×2	5×72×48	ReLU
Conv2D	64@3×3	1×1	3×70×64	ReLU
Conv2D	64@3×3	1×1	2×68×64	ReLU
Flatten	-	-	4352	-
Fully Connected	-	-	100	ReLU
Fully Connected	-	-	50	ReLU
Fully Connected	-	-	10	ReLU
Fully Connected	-	-	1	Tanh

Table 3.5: Overview of the various layers in the SpatialSpurvPilotExtended model, based on Bojarski, Del Testa, et al. (2016) and Hubschneider et al. (2017).

Layer type	Kernels@Kernel size	Stride size	Output size	Activation
Input	-	-	3×600×60	-
Conv2D	24@5×5	2×2	31×298×24	ReLU
Conv2D	36@5×5	2×2	14×147×36	ReLU
Conv2D	48@5×5	2×2	5×72×48	ReLU
Conv2D	64@3×3	1×1	3×70×64	ReLU
Conv2D	64@3×3	1×1	2×68×64	ReLU
Flatten	-	-	4352	-
Fully Connected	-	-	500	ReLU
Fully Connected	-	-	100	ReLU
Fully Connected	-	-	50	ReLU
Fully Connected	-	-	10	ReLU
Fully Connected	-	-	1	Tanh

- A marker turning task on the SPURV robot, where multiple images from different points in time, called a spatial history, are sent into the model to give it an overview of its surroundings

For each task, a neural network is trained on the collected data, while using MSE as the loss. The trained models are evaluated by utilising the quantitative metrics MAE and a whiteness measure that describes the stability of the predicted steering angles. Visualisations of the car's path given the predicted steering angles are generated and used in the evaluation, along with VBP heat maps that show where the network focuses its attention in the input image. Finally, the models are tested in real-life to see how they perform.

Chapter 4

Results

This chapter presents the results obtained through the experiments described in Chapter 3. The results on the SPURV and Udacity simulator lane following tasks are presented in Section 4.1 and Section 4.2, respectively. The marker turning task results are introduced in Section 4.3. For each task, the quantitative results are presented first, followed by the qualitative results and findings from real-life and interactive testing. The chapter is finalised with a summary of the results related to the choice of hyperparameters in Section 4.4.

The videos mentioned in this chapter can be found in the following playlist: `https://goo.gl/XVQvGV`. The individual videos are also linked to in the text when mentioned.

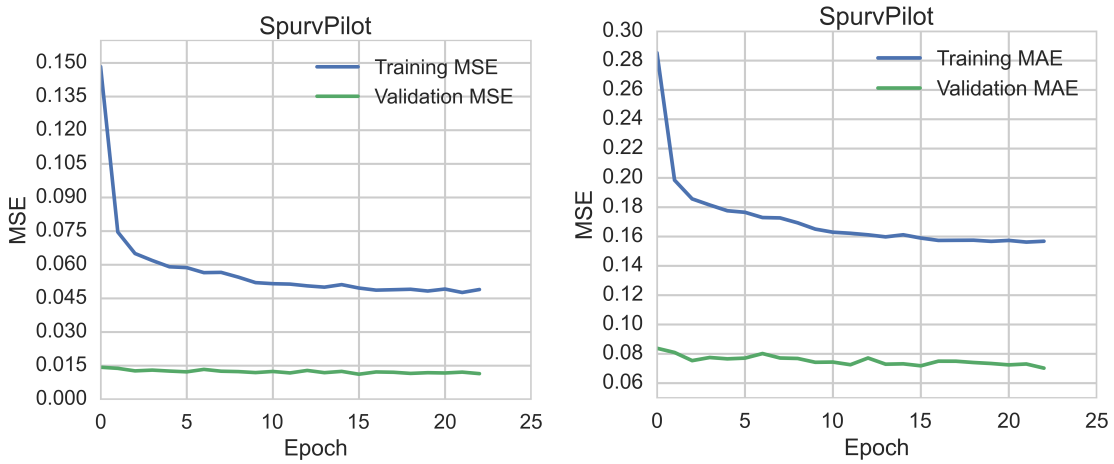
4.1 SPURV Lane Following

The real-life testing of the lane following task using the SPURV was successful. The trained *SpurvPilot* model was able to follow indoor lanes with no particular problems. The results are presented in this section, starting with the quantitative results before moving on to the qualitative ones. Finally, the results from the real-life test are summarised.

4.1.1 Quantitative Metrics

The final *SpurvPilot* model had a MSE of 0.0489 on the training set, 0.0114 on the validation set and 0.0227 on the test set. The complete quantitative results are listed in Table 4.1. The training and validation MSE loss plots along with the MAE plots for the final *SpurvPilot* model can be seen in Figure 4.1.

A visualisation of the stability of the predictions, i.e. the whiteness over time for the predicted steering angles in the test set, can be seen in Figure 4.2(a). The whiteness of the ground truth angles is seen in Figure 4.2(b). The average whiteness of the predictions on the test set was 0.0003 rad^2/frame, which corresponds to an average difference in steering angles of 0.017 rad between each image frame. In comparison, the whiteness of the ground truth steering angles was larger, at 0.0023 rad^2/frame.



(a) **(b)**

Figure 4.1: Training history plots showing the development in MSE and MAE during training for the SPURV lane following task.

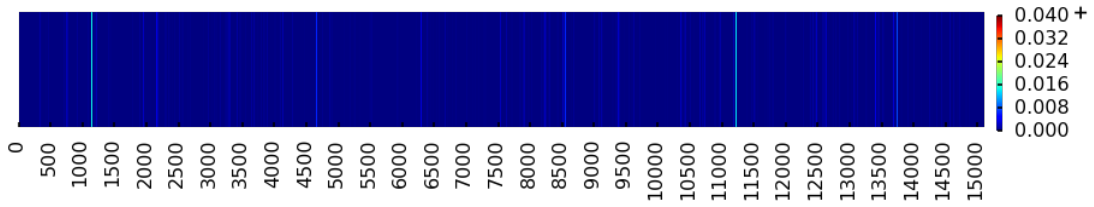
Table 4.1: The quantitative results of *SpurvPilot* on the SPURV lane following task.

	Training	Validation	Testing
MSE	0.0489	0.0114	0.0227
MAE	0.1568	0.0702	0.1018
Whiteness [rad^2/frame]	-	0.0002	0.0003
Whiteness [deg^2/frame]	-	0.6089	0.9208

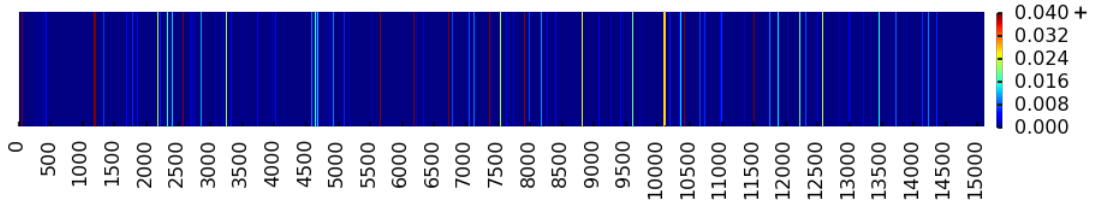
4.1.2 Car Path Visualisation

For each image, an estimated driving path over the succeeding 30 frames was calculated and overlayed onto the image. Figure 4.3 shows samples from the test set of the predicted driving path of the *SpurvPilot* model, compared to the ground truth path.

The predicted driving paths generally show satisfactory behaviour. A small part of the images show a perfect or near-perfect match between the target and predicted paths as in Figure 4.3(a) and (b). The driving paths have varying degrees of divergence from the ground truth, as shown in Figure 4.3(c), (d), (e) and (f), but mostly the



(a) Prediction whiteness



(b) Ground truth whiteness

Figure 4.2: A visualisation of the stability of the predictions, i.e. the whiteness of the predicted steering angle for each image in the test set (a), compared to the ground truth whiteness (b). The whiteness is measured in rad^2/frame.

driving paths appear to stay inside the lane. No disastrous driving paths were found, but some irregularities are present, as in (f).

4.1.3 Activation Heat Maps

Various examples of VBP heat maps of the activation in the convolutional layers of the *SpurvPilot* network can be seen in Figure 4.4. The samples are taken from the test set.

Figure 4.4(a) and (b) show examples of images where the network focuses on the desired features when driving straight ahead and while in a steep, right turn. This was the case for a fair portion of the test set. In Figure 4.4(c), the network focuses mostly on only one of the ropes, which occurred quite often. A large part of the images show some level of focus on other features in the image, as shown in Figure 4.4(d), (e) and (f). In most cases, the network focuses heavily on the ropes, while also looking at one or two unrelated spots in the image, as in Figure 4.4(d). Background lines that look similar to ropes are also often in focus, as in Figure 4.4(e). Figure 4.4(f) shows an example where the network almost completely ignores the ropes and focuses on the benches in the background instead. This is the case for very few of the images.

4.1.4 Real-life Testing on the SPURV

Real-life testing of the *SpurvPilot* model was conducted in *Location B*, the same room as the majority of training and validation data. The model had no problems

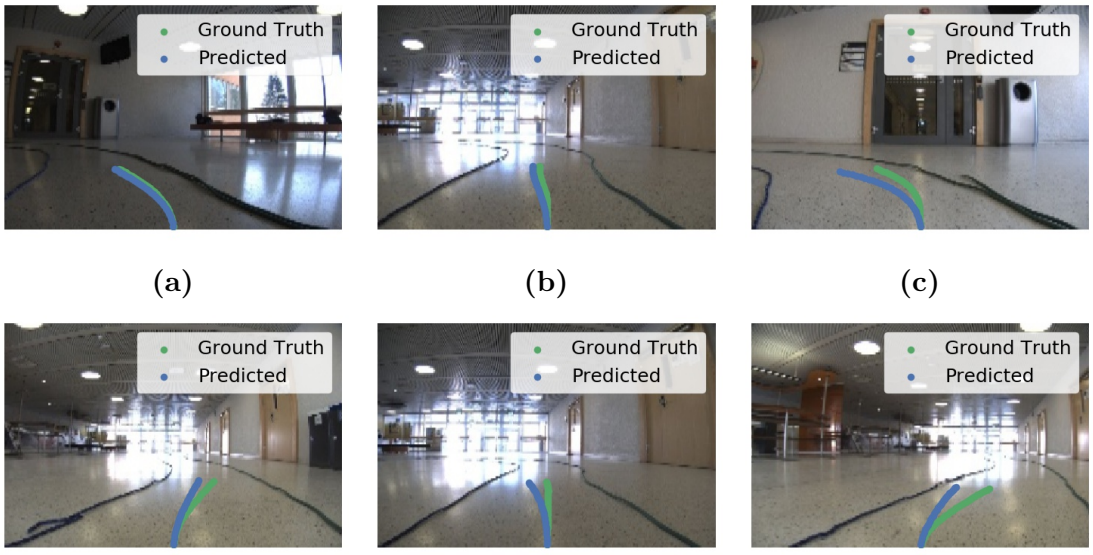


Figure 4.3: Visualisations of predicted driving paths, compared to ground truth paths, on a selection of images from the test set.

following the lanes throughout the entire track, driving both clockwise and counter-clockwise in the track. A selection of images from the real-life testing are shown in Figure 4.5. Videos of this real-life test are available at:

- External camera:
  Title: Real-life test of *SpurvPilot* model in "Location B" (External camera)
  URL: `https://youtu.be/K-yPiKRwxkg`
- Internal camera with heat maps and path visualisation:
  Title: Real-life test of *SpurvPilot* model in "Location B" (Internal camera)
  URL: `https://youtu.be/YGw8MgCGONw`

The *SpurvPilot* model was also tested in a smaller room that was not present in either the training or validation set. It appeared to have no problems following the track. A selection of images from this test can be seen in Figure 4.6. The videos from this test are available at:

- External camera:
  Title: Real-life test of *SpurvPilot* model in "Location D" (External camera)
  URL: `https://youtu.be/iE_DRgRSAGc`
- Internal camera with heat maps and path visualisation:
  Title: Real-life test of *SpurvPilot* model in "Location D" (Internal camera)
  URL: `https://youtu.be/nk-eqKWDQj8`

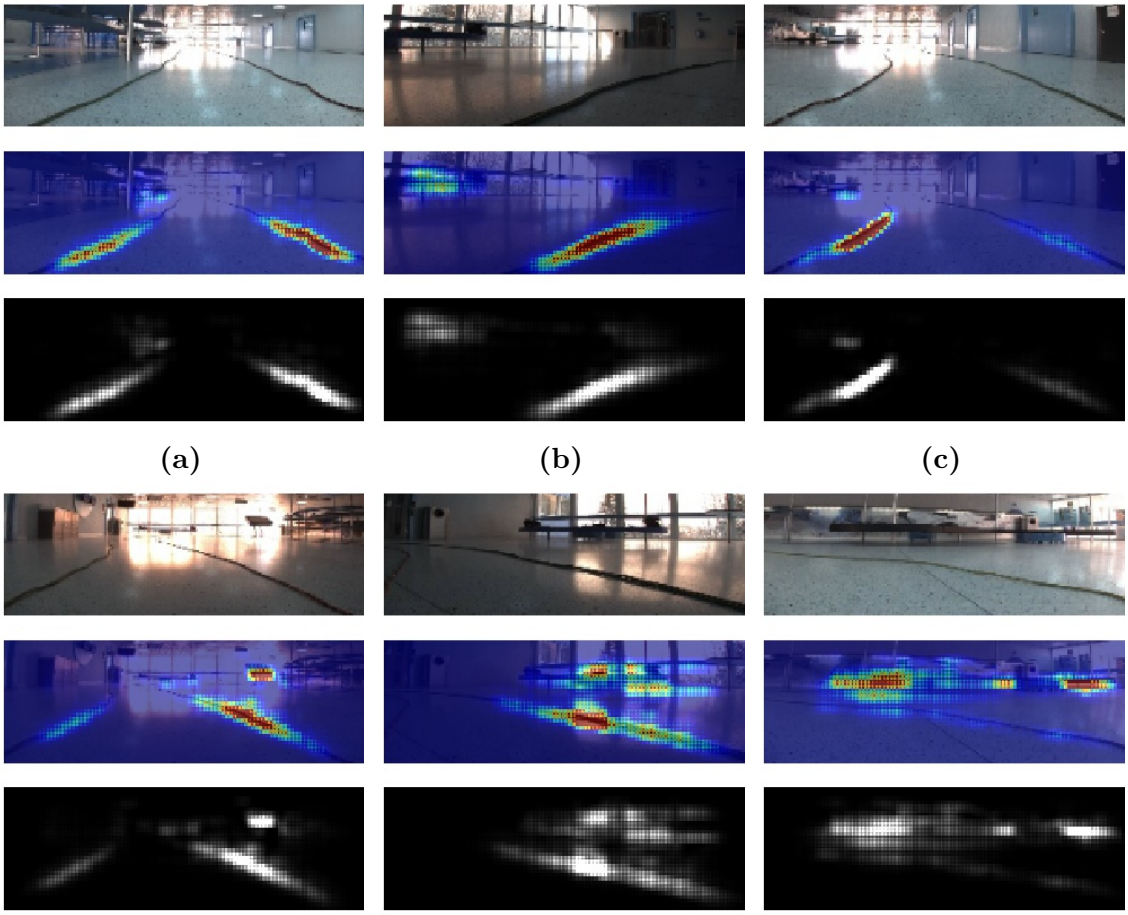
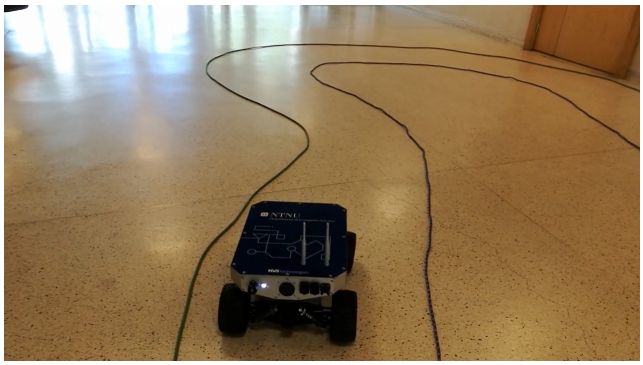


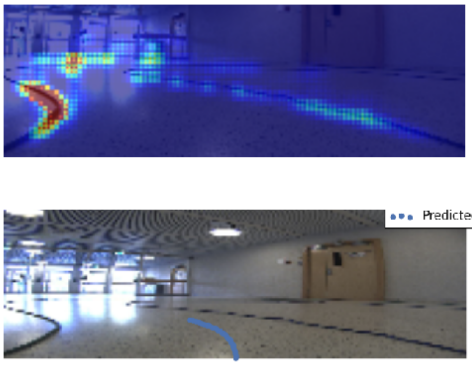
Figure 4.4: The *SpurvPilot*'s VBP heat maps on a selection of images from the test set. The bottom row shows examples where the model focuses on undesired features in the image.

A number of edge cases were also recorded. The edge cases included a very steep turn the SPURV would barely be able to make, another steep turn that was impossible for the SPURV to make, and a case of straightening up when partly outside of the tracks. The three situations are pictured in Figure 4.7. The *SpurvPilot* failed at the steepest turn, but succeeded at all of the other edge cases. A video of the edge cases is available at:

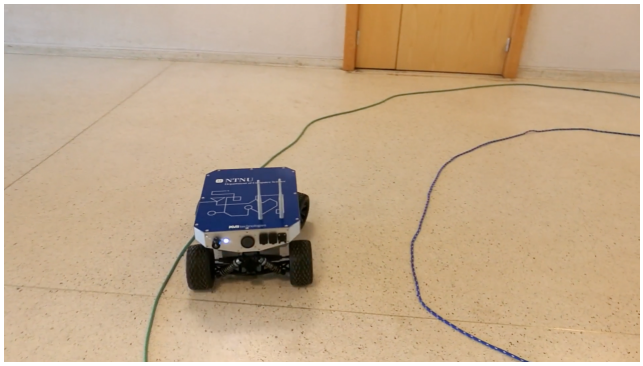
- External and internal camera:
  Title: *SpurvPilot* Edge Cases
  URL: `https://youtu.be/0RgF0_HRV70`



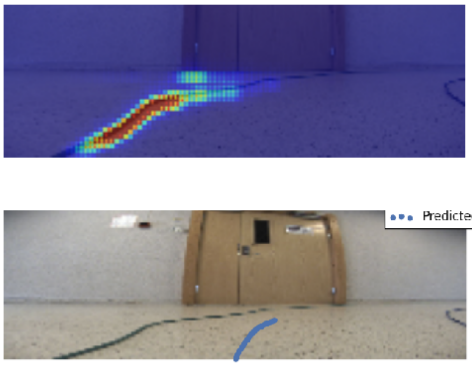
(a) External camera video screenshot before turn



(b) Internal camera video screenshot before turn, with path and heat map



(c) External camera screenshot during turn

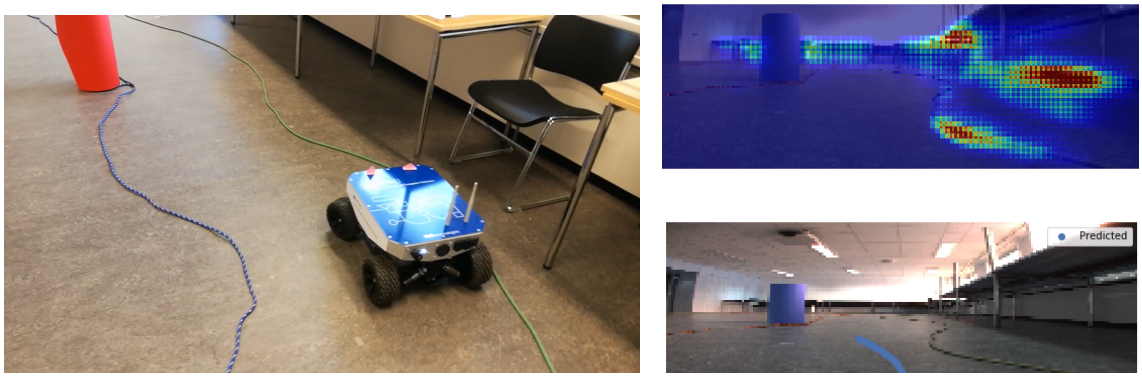


(d) Internal camera video screenshot during turn, with paths and heat maps

Figure 4.5: Screenshots of the videos from the real-life testing of *SpurvPilot* in *Location B*.

4.1.5 Prediction Time

When running the *SpurvPilot* model on the SPURV's NVIDIA Jetson TX2 GPU, the network was able to perform an average of 62 predictions per second.



(a) External camera video screenshot

(b) Internal camera video screenshot with overlayed paths and heat maps

Figure 4.6: Screenshots of the real-life testing of *SpurvPilot* in *Location D*.

4.2 Udacity Simulator Lane Following

This section covers the results of the lane following task in the Udacity Simulator. The $SpurvPilot_{Udacity}$ model was initialised with the weights from the *SpurvPilot* and fine-tuned on data collected from the Udacity Simulator. The trained model was able to follow the Lake Track in the simulator. The results are presented in this section, starting with the quantitative metrics before summarising the qualitative results along with the observations from the final testing in the simulator.

4.2.1 Quantitative Metrics

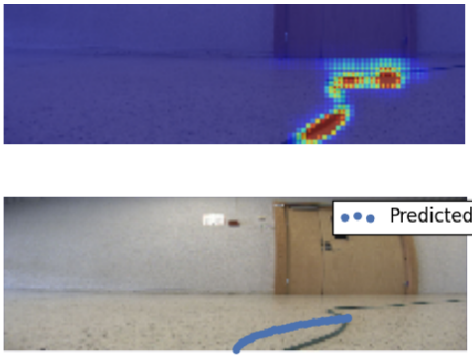
The final $SpurvPilot_{Udacity}$ model had a MSE of 0.0283 on the training set, 0.0066 on the validation set and 0.0132 on the test set. The complete quantitative results are listed in Table 4.2. The training and validation MSE loss plots along with the MAE plots for the final *SpurvPilot* model can be seen in Figure 4.8.

A visualisation of the stability of the predictions, i.e. the whiteness over time for the predicted steering angles in the test set, can be seen in Figure 4.9(a). The average whiteness of the ground truth angles is seen in Figure 4.9(b). The whiteness of the predictions on the test set was 0.0007 rad^2/frame, which corresponds to an average difference in the steering angle of 0.0264 rad between individual frames. In comparison, the whiteness of the ground truth steering angles was 0.0057 rad^2/frame.

4.2.2 Activation Heat Maps

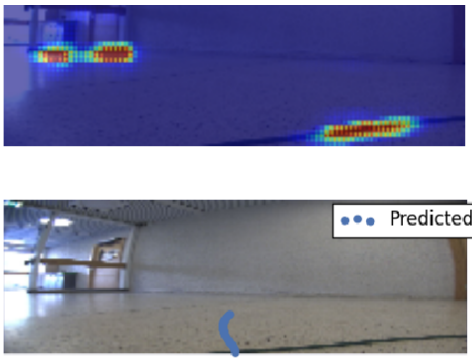
A selection of heat maps from the test set of the Udacity Simulator task can be seen in Figure 4.10. From the images, it can be seen that the model focuses both on the

(a) External camera video screenshot during very steep turn

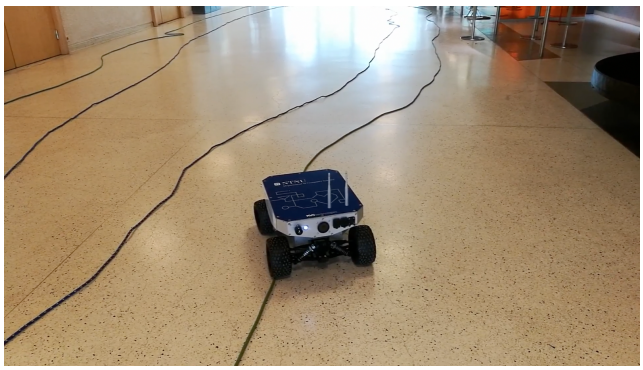


(b) Internal camera video screenshot with paths and heat maps during very steep turn

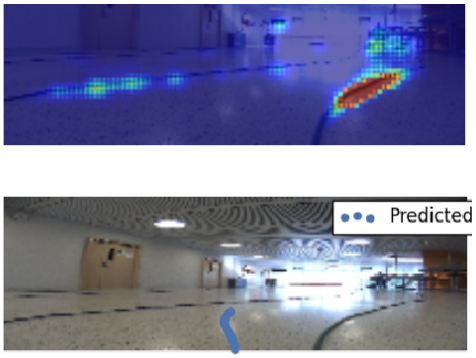
(c) External camera screenshot during physically impossible turn, just before the SPURV loses sight of the rope ahead of it



(d) Internal camera video screenshot with paths and heat maps during physically impossible turn, just before the SPURV loses sight of the rope ahead of it



(e) External camera screenshot of the SPURV outside of the tracks

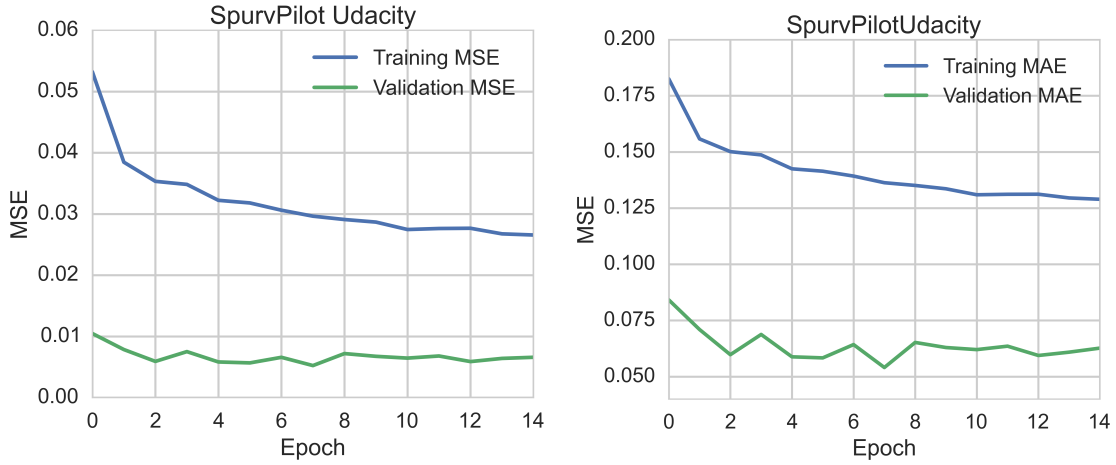


(f) Internal camera video screenshot with paths and heat maps from the SPURV before straightening up inside the tracks

Figure 4.7: Screenshots of the edge cases in the videos from the real-life testing of *SpurvPilot* in *Location B*.

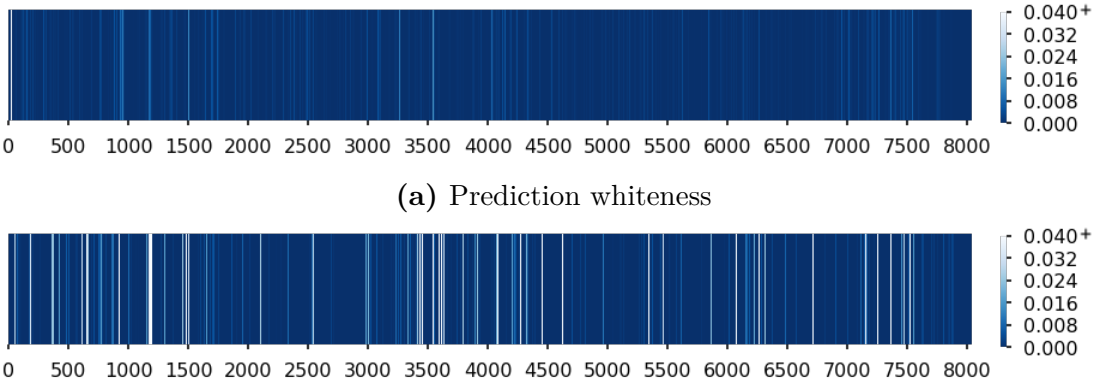
Table 4.2: The quantitative results of $SpurvPilot_{Udacity}$ model.

	Training	Validation	Testing
MSE	0.0283	0.0066	0.0132
MAE	0.1289	0.0627	0.0857
Whiteness [rad^2/frame]	-	0.0003	0.0007
Whiteness [deg^2/frame]	-	1.000	2.458



(a) **(b)**

Figure 4.8: Training history plots showing the development in MSE and MAE during training of the $SpurvPilot_{Udacity}$ model.



(a) Prediction whiteness

(b) Ground truth whiteness

Figure 4.9: A visualisation of the whiteness of the predicted steering angle for each image in the Udacity Simulator test set (a), compared to the ground truth whiteness (b). The whiteness is measured in rad^2/frame.

outlines of the road and other features in the images, like the horizon ((b) and (c)) and the landscape close to the road ((d) and (f)).

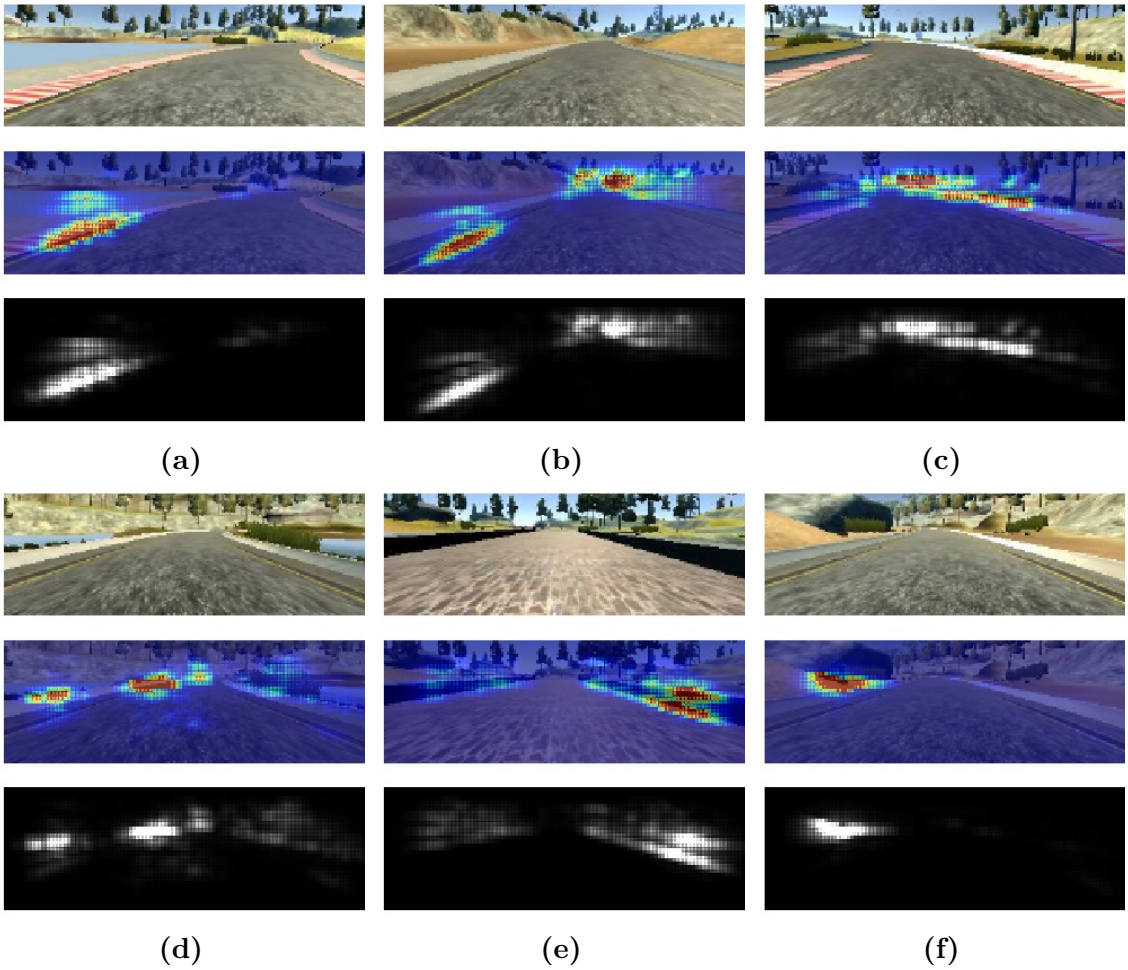


Figure 4.10: VBP heat maps of a selection of images from the validation set on the $SpurvPilot_{Udacity}$ model.

4.2.3 Simulation Tests

The $SpurvPilot_{Udacity}$ model was tested in the Udacity Simulator. It was able to follow the track with almost no problems, only once being a bit close to the edge during a steep turn. A video of the test with VBP heat maps can be found here:

- External and internal camera:
  Title: Simulator test of $SpurvPilot_{Udacity}$ on the Lake Track
  URL: `https://youtu.be/Zcjvi7_Fmqs`

4.3 Marker Turning Task

This section presents the results of the two architectures *SpatialSpurvPilot* and *SpatialSpurvPilotExtended*. Both models were partially able to complete the marker turning task during real-life testing but displayed unstable behaviour. The quantitative results are presented first, before the qualitative results are summarised. The results from real-life testing are shown at the end of this section.

4.3.1 Quantitative Metrics

The final *SpatialSpurvPilot* model had a MSE of 0.0278 on the training set and 0.0710 on the validation set. The complete quantitative results are listed in Table 4.3. The training and validation MSE loss plots along with the MAE plots for the final *SpatialSpurvPilot* model can be seen in Figure 4.11. The corresponding plots for *SpatialSpurvPilotExtended* are shown in Figure 4.12.

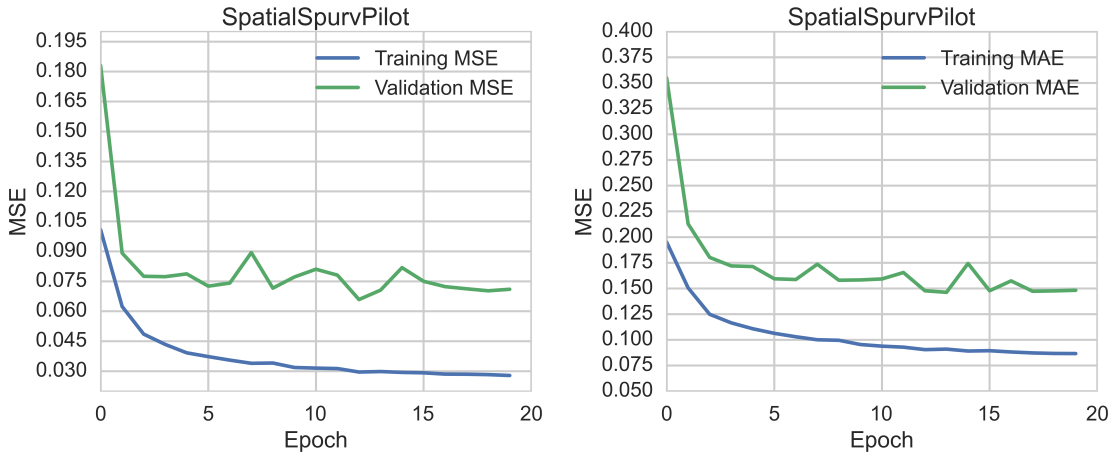
A plot of the whiteness of the *SpatialSpurvPilot* model on the entire validation set is shown in Figure 4.13, while Figure 4.14 presents the corresponding plot for *SpatialSpurvPilotExtended*.

Table 4.3: The quantitative training and validation results of the *SpatialSpurvPilot* and *SpatialSpurvPilotExtended* models on the marker turning task.

	SpatialSpurvPilot		*SpatialSpurvPilotExtended*	
	Training	Validation	Training	Validation
MSE	0.0278	0.0710	0.0273	0.0762
MAE	0.0865	0.1481	0.0975	0.1442
Whiteness [rad^2/frame]	-	0.0073	-	0.0071
Whiteness [deg^2/frame]	-	23.9532	-	23.3407

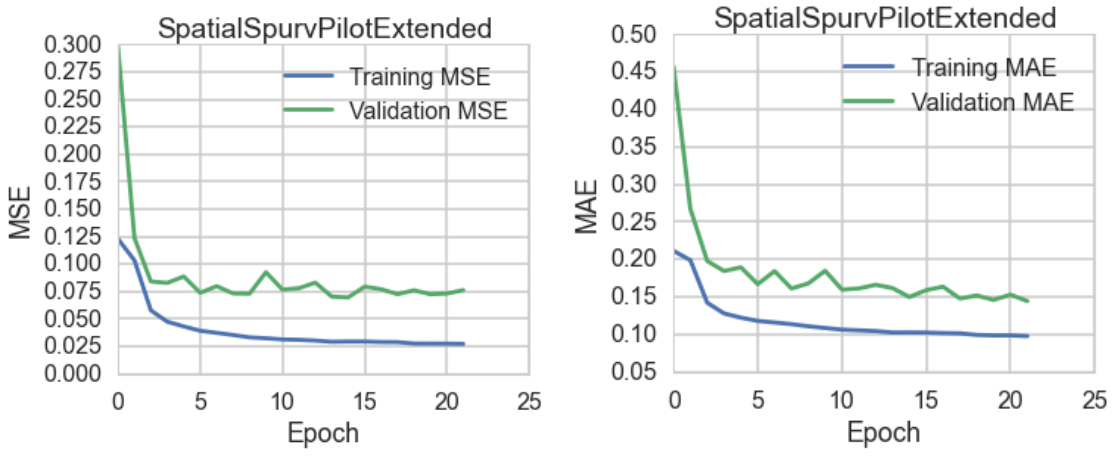
4.3.2 Car Path Visualisation

A selection of the generated path visualisations for *SpatialSpurvPilot* are shown in Figure 4.15. Figure 4.15(a) indicates that the model tends to turn slightly earlier to the right than the ground truth, the model also seems to terminate the right turn a while before the ground truth, as seen in Figure 4.15(b). The full left turns tend to be identical to the ground truth, shown in Figure 4.15(c). Sometimes the model does not drive towards a marker ahead of it, especially if the marker is relatively far ahead as in Figure 4.15(d). Visualisations for *SpatialSpurvPilotExtended* are not included here, as they do not show any different behaviour from *SpatialSpurvPilot*.



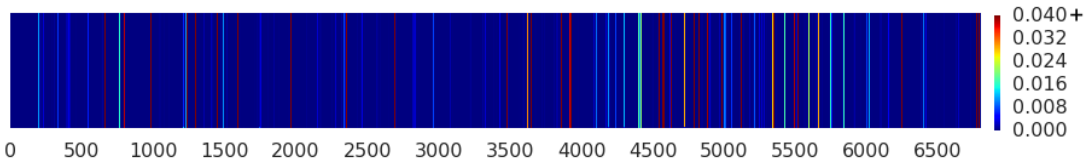
(a) **(b)**

Figure 4.11: Training history plots showing the development in MSE and MAE during training of *SpatialSpurvPilot* for the marker turning task.

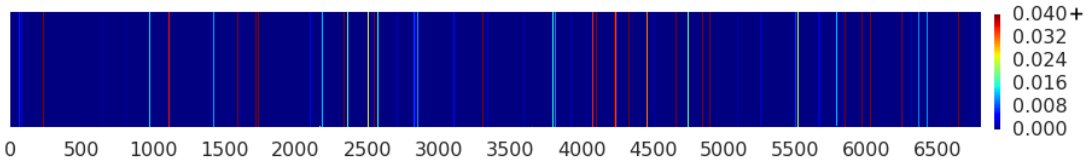


(a) **(b)**

Figure 4.12: Training history plots showing the development in MSE and MAE during training of *SpatialSpurvPilotExtended* for the marker turning task.

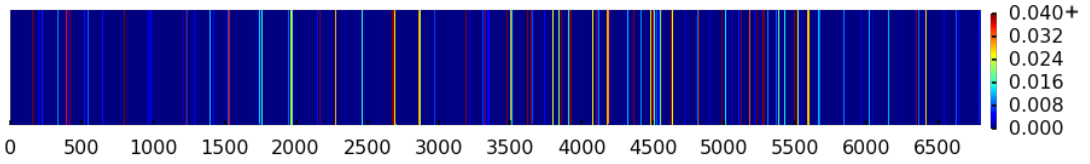


(a) Prediction whiteness

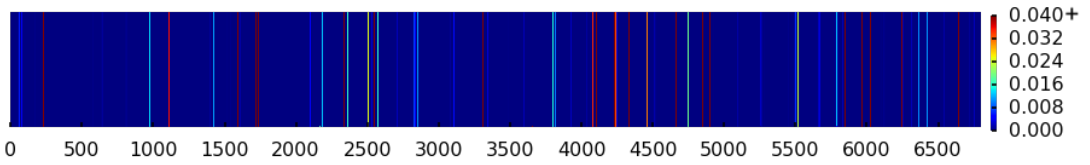


(b) Ground truth whiteness

Figure 4.13: A visualisation of the whiteness of the predicted steering angle by *SpatialSpurvPilot* for each image in the validation set (a), compared to the ground truth whiteness (b). The whiteness is measured in rad^2/frame.



(a) Prediction whiteness



(b) Ground truth whiteness

Figure 4.14: A visualisation of the whiteness of the predicted steering angle by *SpatialSpurvPilotExtended* for each image in the validation set (a), compared to the ground truth whiteness (b). The whiteness is measured in rad^2/frame.

The reader is encouraged to watch the video below for visualisations on a part of the validation set for both models:

- Internal camera with heat maps and path visualisation:
  Title: Path Visualisation and VBP Heat Maps on Part of Validation Set
  URL: `https://youtu.be/UTvfqDVgJb4`

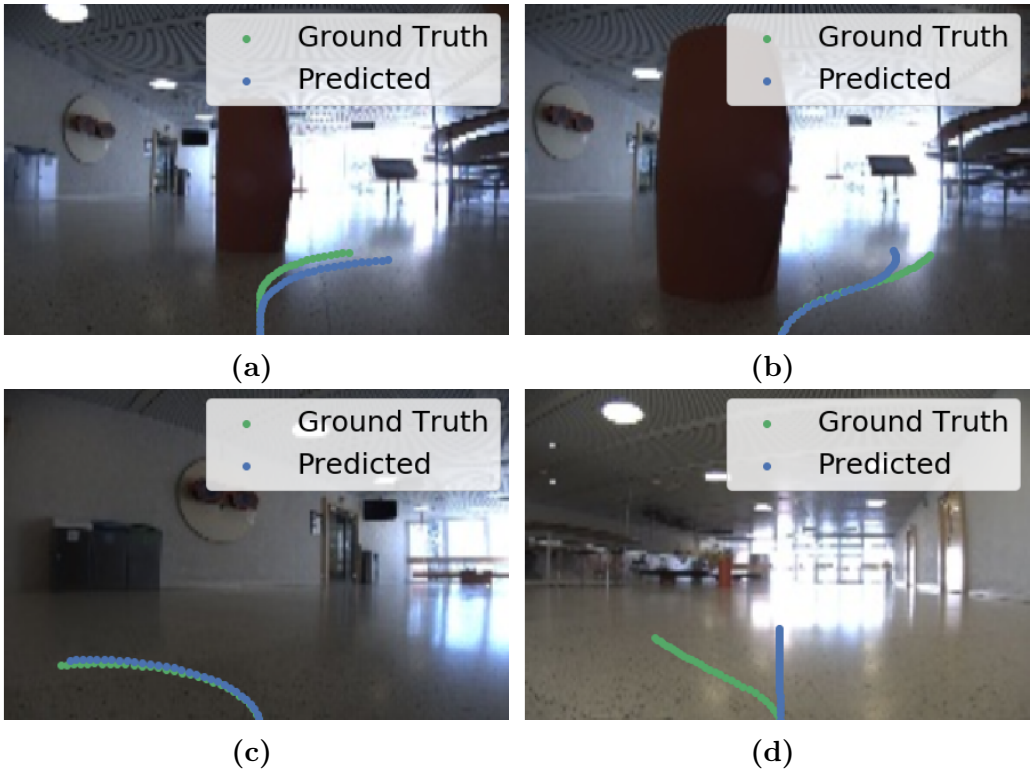
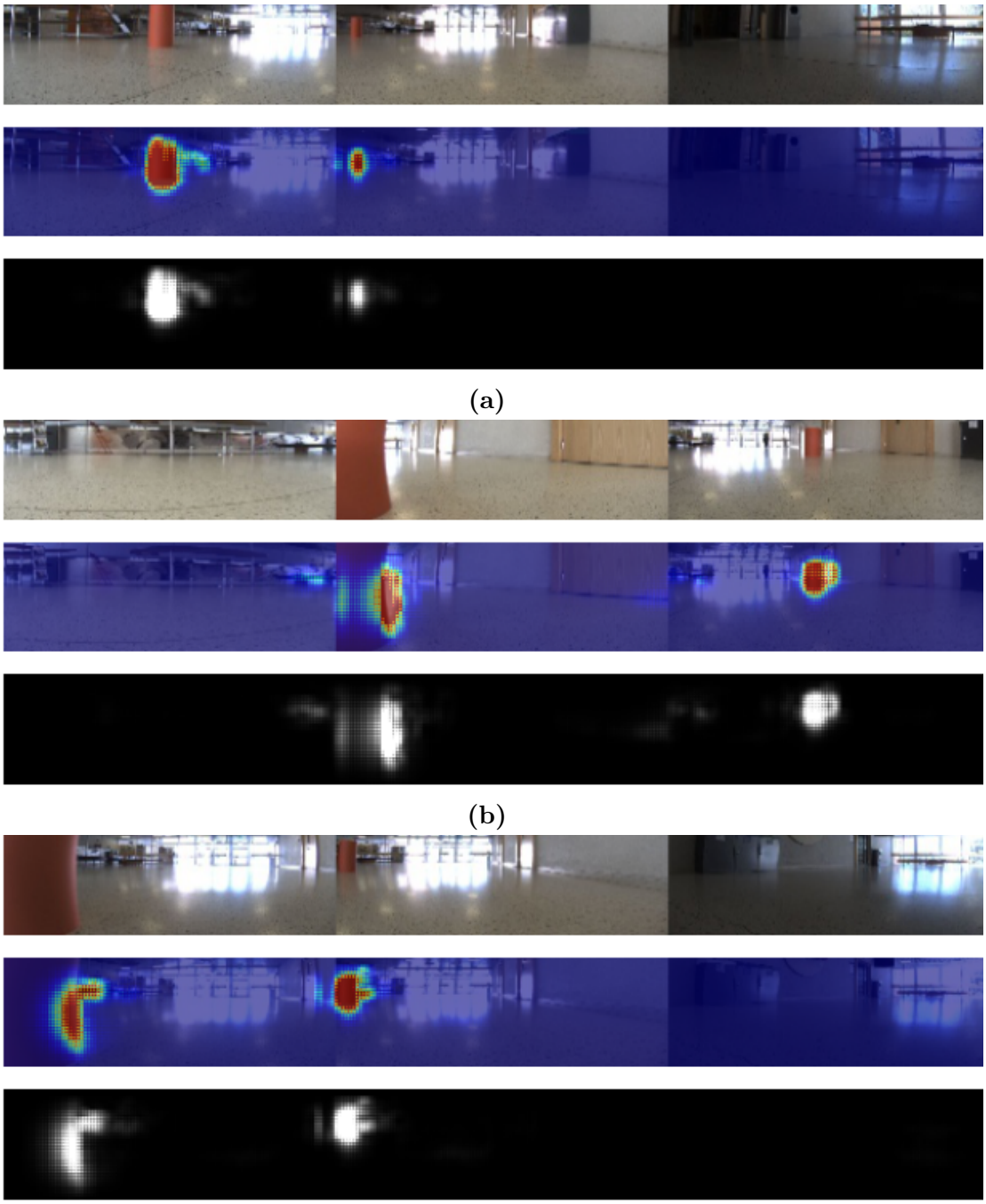


Figure 4.15: Visualisations of predicted driving paths by *SpatialSpurvPilot*, compared to ground truth paths, on a selection of images from the validation set for the marker turning task.

4.3.3 Activation Heat Maps

Figure 4.16 shows a selection of VBP heat maps of *SpatialSpurvPilot* where the model seems to correctly focus on the markers in spatial history. The corresponding heat maps of *SpatialSpurvPilotExtended* can be seen in Figure 4.17, and show that the extended model seems to struggle with focusing on only the markers.

Figure 4.18(a) and Figure 4.18(b) show what the *SpatialSpurvPilot* model focuses on when given a spatial history where no markers are present. Figure 4.18(c) shows a heat map where the model seems to struggle with spotting the markers when they stand in heavy backlight.

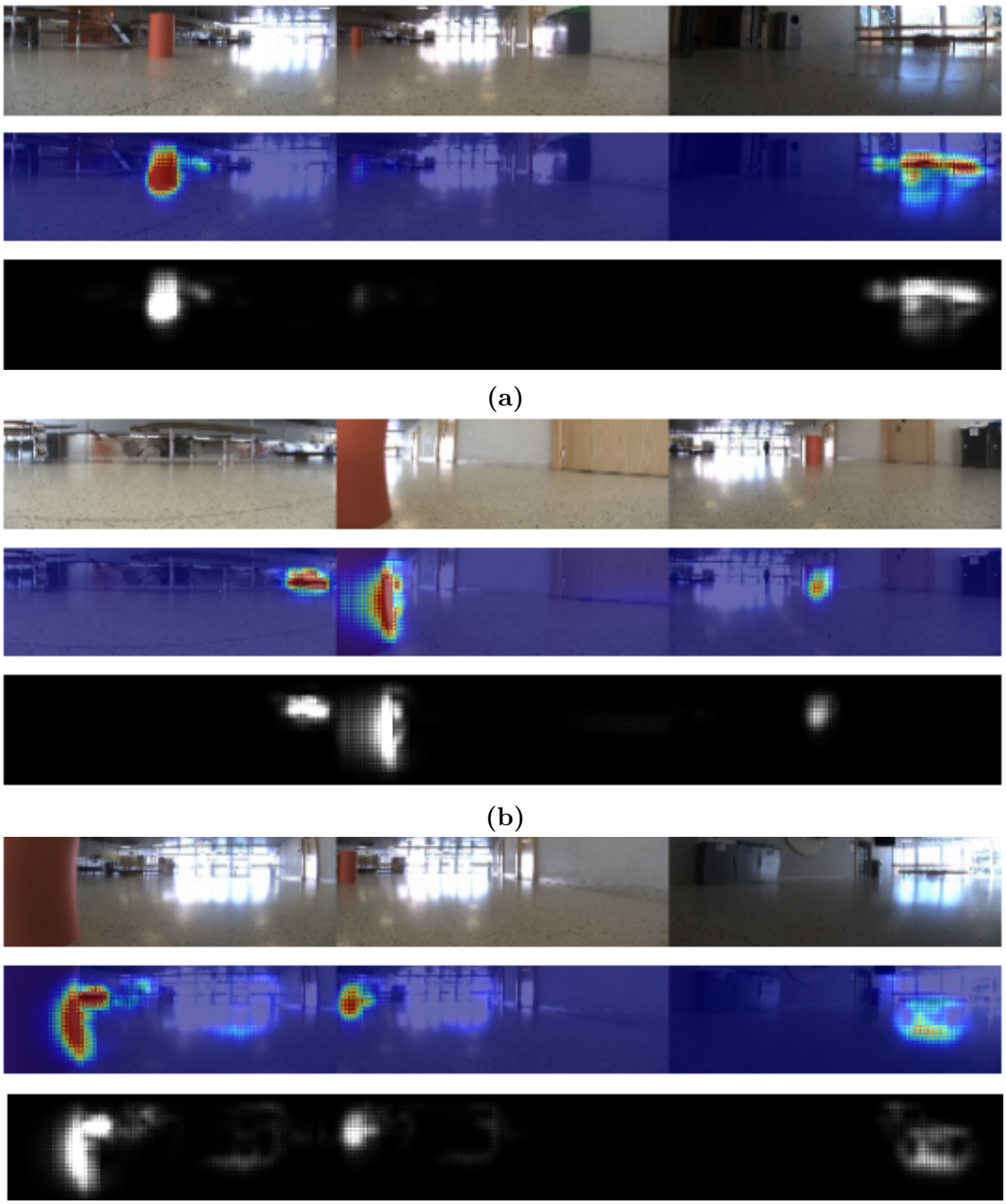


(a)

(b)

(c)

Figure 4.16: VBP heat maps for *SpatialSpurvPilot* trained on the marker turning task. The leftmost images are the most recent in the spatial history, and the following images are 80 frames apart, as explained in Figure 3.22.

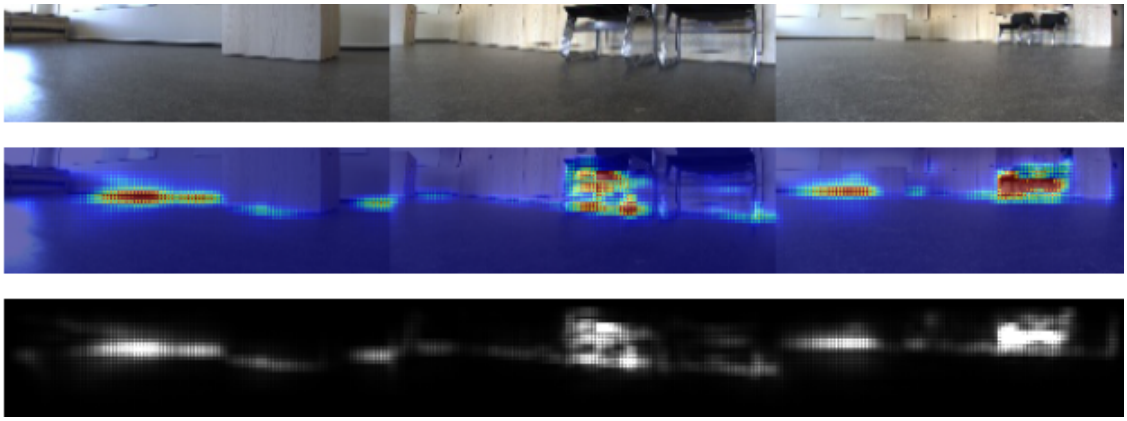


(a)

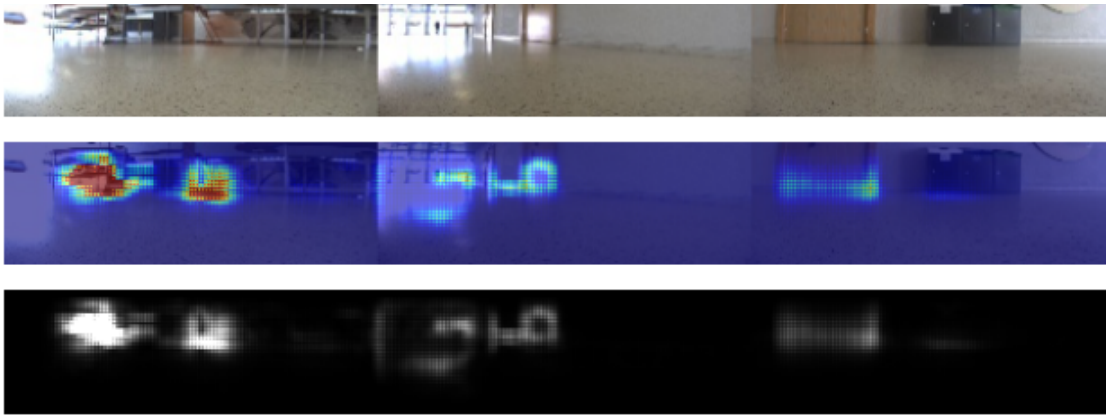
(b)

(c)

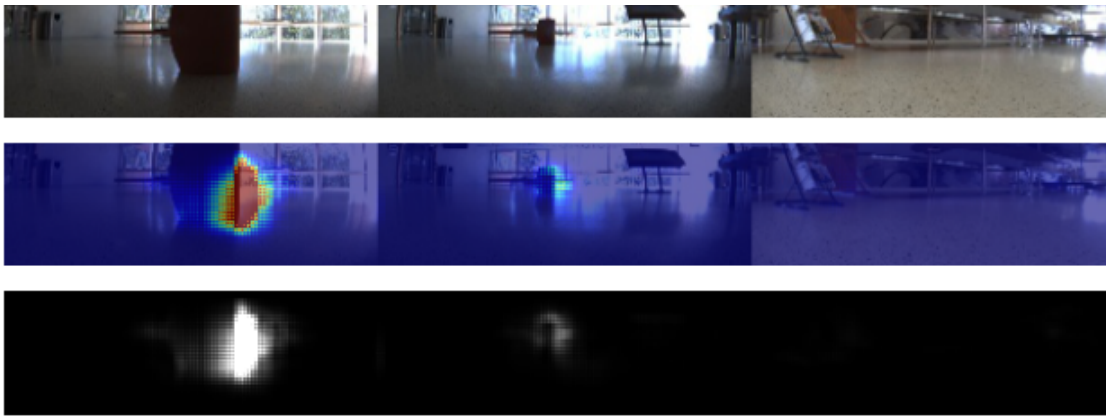
Figure 4.17: VBP heat maps for *SpatialSpurvPilotExtended* trained on the marker turning task. The leftmost images are the most recent in the spatial history, and the following images are 80 frames apart, as explained in Figure 3.22.



(a) No markers in spatial history



(b) No markers in spatial history



(c) Marker detection issues in images with backlight

Figure 4.18: VBP heat maps for *SpatialSpurvPilot* trained on the marker turning task. The leftmost images are the most recent in the spatial history, and the following images are 80 frames apart, as explained in Figure 3.22.

4.3.4 Real-life Testing on the SPURV

A real-life test of the *SpatialSpurvPilot* and *SpatialSpurvPilotExtended* models was conducted in *Location B*. Both models showed unstable behaviour, often failing to drive towards a marker straight ahead of the SPURV. Putting the markers in less challenging lighting conditions seemed to help both models with this issue. Nevertheless, both models showed undesired behaviours. These are compiled in Figure 4.19 and briefly described below.

Both models terminated their action of turning right slightly too early, causing the side of the SPURV to brush up against the marker when turning around it, as seen in Figure 4.19(a).

Another problem that became apparent during real-life testing, was that both models had a tendency to drive straight for a portion of time, although there were markers present in their spatial history, as shown in Figure 4.19(d). This again caused the turns to become sub-optimal and more time-consuming than what the intention was when approximating the D_f value in Section 3.7.2. Thus, the spatial history wrongfully ended up without any markers in certain situations, like the one in Figure 4.19(f), causing the SPURV to drive straight - an action that was right given the model's spatial history, but wrong considering its external surroundings.

During the real-life test *SpatialSpurvPilotExtended* appeared to show more stable behaviour than *SpatialSpurvPilot*, reflected in their respective whiteness values of 0.0081 and 0.0100 rad^2/frame on the predictions during the real-life test.

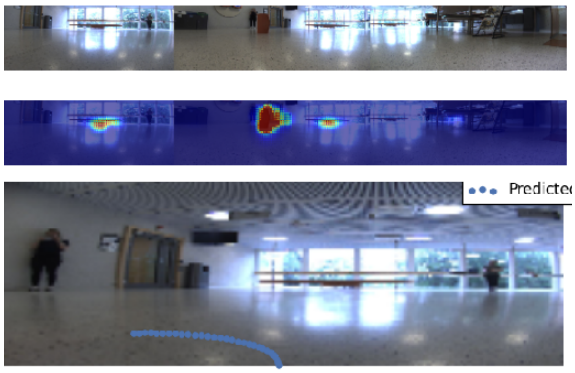
- External and internal camera with heat maps and path visualisation:
  Title: Real-life test of *SpatialSpurvPilot* model in "Location B"
  URL: `https://youtu.be/rmeffyZEJ38`

- External and internal camera with heat maps and path visualisation:
  Title: Real-life test of *SpatialSpurvPilotExtended* model in "Location B"
  URL: `https://youtu.be/yfq45hRYx_k`

4.3.5 Prediction Time

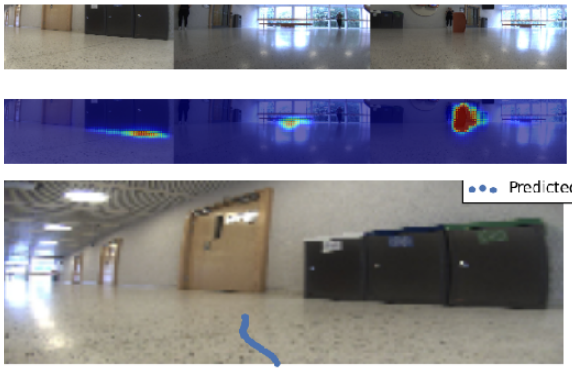
When running the *SpatialSpurvPilot* model on the SPURV's NVIDIA Jetson TX2 GPU, the network was able to perform predictions at a rate of 47 fps. The *SpurvPilotExtended* predicted steering angles at an average rate of 37 fps.

(a) External camera video screenshot showing the SPURV brushing up against the marker



(b) Internal camera video screenshot with paths and heat maps when the SPURV brushes up against marker

(c) External camera screenshot as SPURV drives straight when it is not supposed to



(d) Internal camera video screenshot with paths and heat maps as the SPURV drives straight even though it has a marker in its spatial history

(e) External camera screenshot of the SPURV driving straight although there are markers around

(f) Internal camera video screenshot with paths and heat maps where the model has no markers in its spatial history because of an ineffective turn

Figure 4.19: Screenshots of sub-optimal behaviour during real-life testing of *SpatialSpurvPilot* in *Location B*.

4.4 Hyperparameters

This section summarises the most important findings during hyperparameter tuning, clarifying why they were set to the values described in Section 3.3.

4.4.1 Optimisers

Several different combinations of optimisers and learning rates were assessed for the training of the *SpurvPilot* model. A selection of the most interesting findings is shown in Figure 4.20. The four optimisers with the most reasonable training history plots were further investigated by inspecting the VBP heat maps. The general trend, especially in challenging light conditions, is illustrated in Figure 4.21. From this, Adam with a learning rate of 0.0003 was chosen.

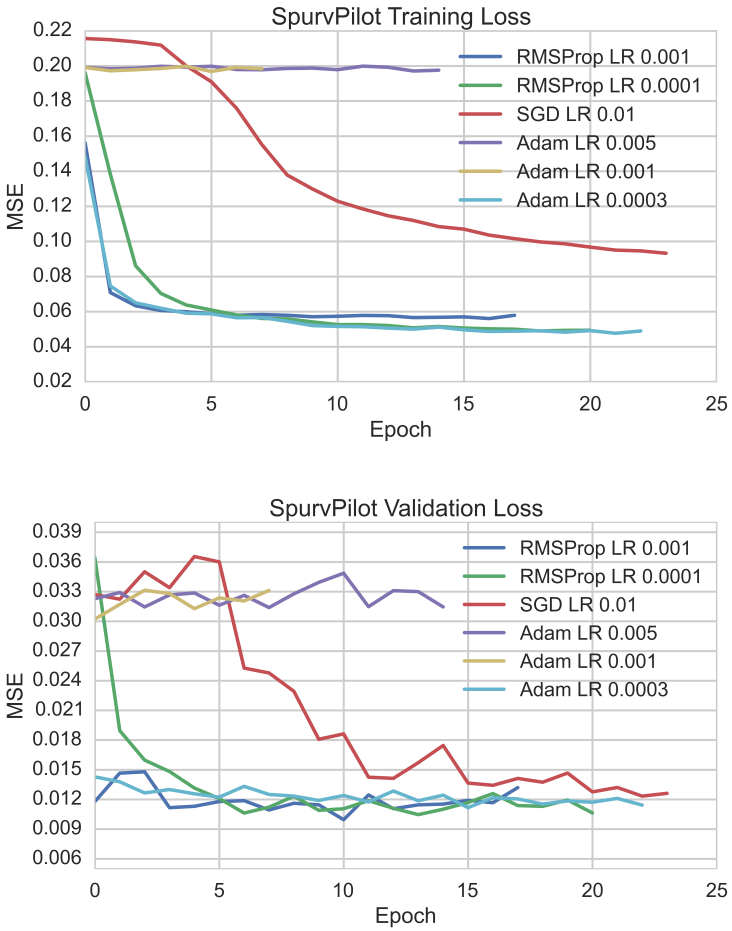
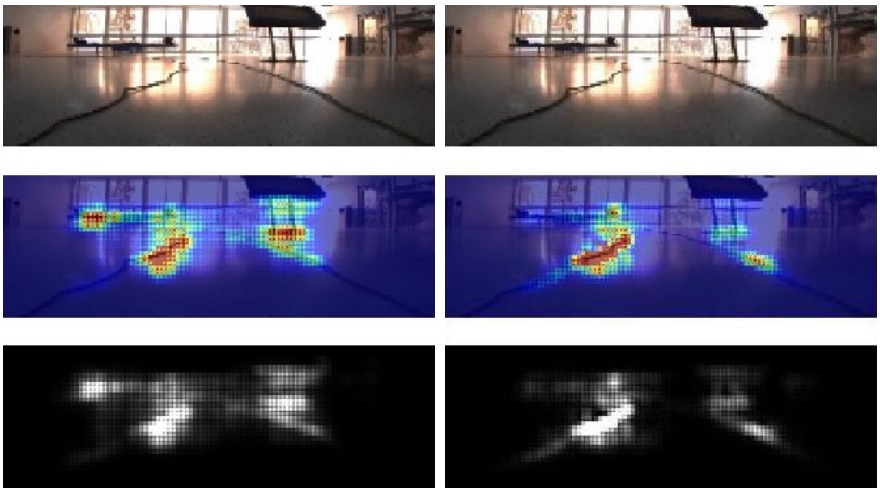
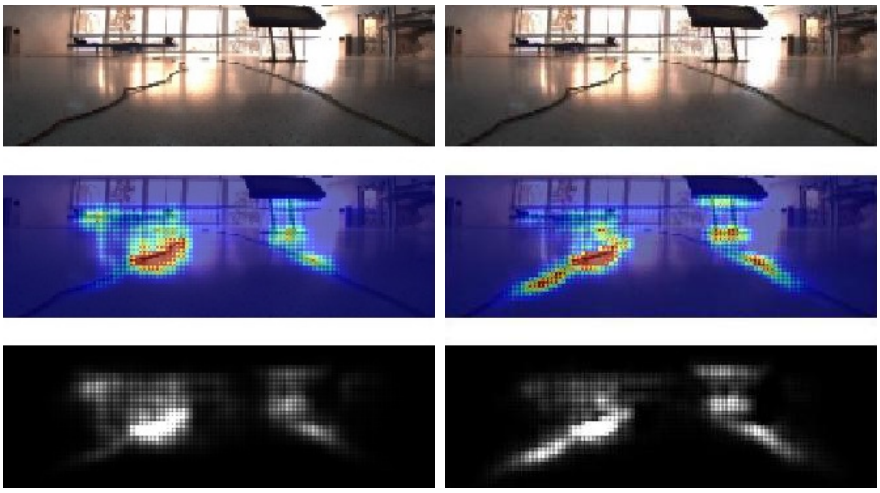


Figure 4.20: Training (top) and validation loss (bottom) of the *SpurvPilot* model trained with various optimisers and learning rates (LR).



(a) SGD LR 0.01 **(b)** RMSProp LR 0.001

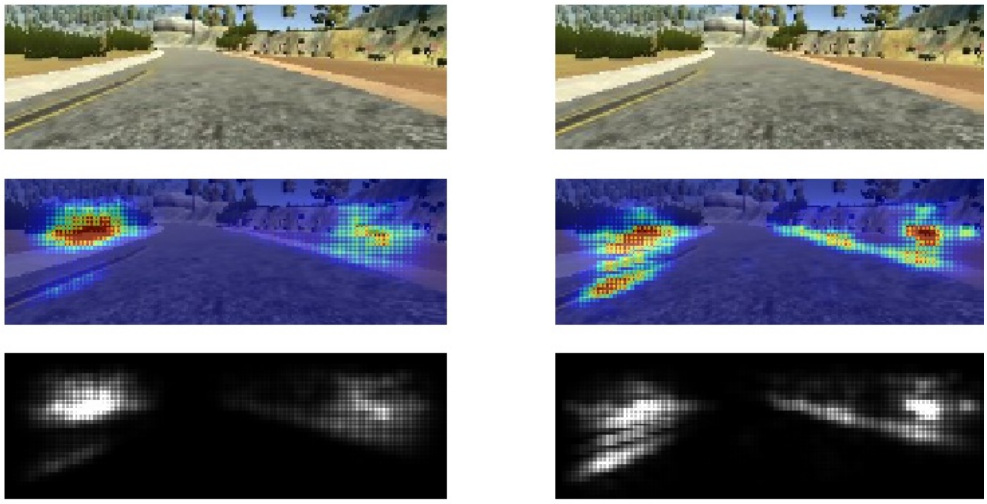


(c) RMSProp LR 0.0001 **(d)** Adam LR 0.0003

Figure 4.21: VBP heat maps from the *SpurvPilot* model trained with various optimisers and learning rates (LR).

4.4.2 Pretraining

The Udacity Simulator model was pretrained on the SPURV indoor lane following data. The resulting heat maps showed that the pretrained model focused more on the outlines of the road, compared to a randomly initialised model. An example can be seen in Figure 4.22



(a) Without pretraining **(b)** With pretraining

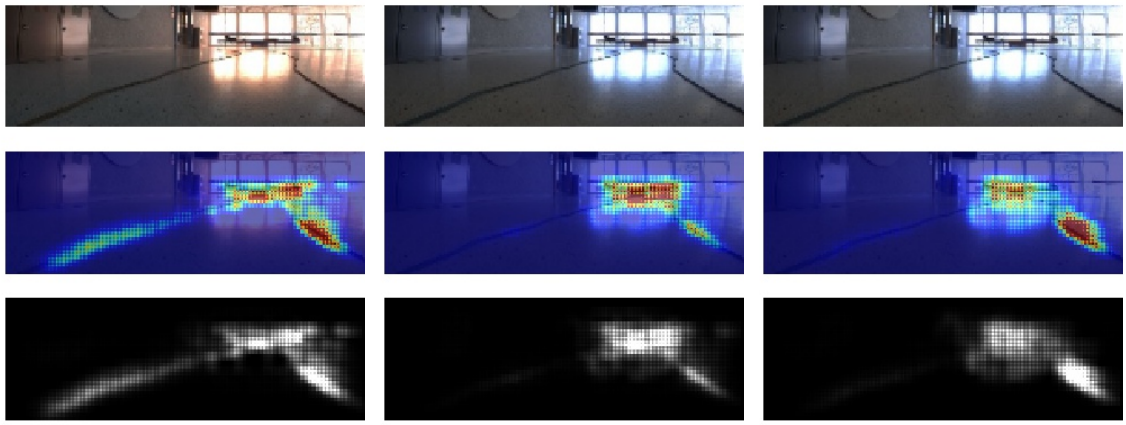
Figure 4.22: VBP heat maps from the $SpurvPilot_{Udacity}$ model with and without pretraining.

4.4.3 Data Augmentation

Figure 4.23 shows a selection of VBP heat maps from models with different combinations of augmentation techniques in images with backlight and glare. The final *SpurvPilot* model, as seen in Figure 4.23(a) and (d), keeps some focus on the lanes ahead. Simultaneously, the same model trained without random erasing seems to lose focus on the lanes. Instead, it appears to focus more on the bright spots in the windows behind, as seen in Figure 4.23(b) and Figure 4.23(e). Figure 4.23(c) and (f) show the heat maps of the model that in addition to no random erasing, also is trained without HSV adjustment.

Horizontal Shifts and Side Cameras

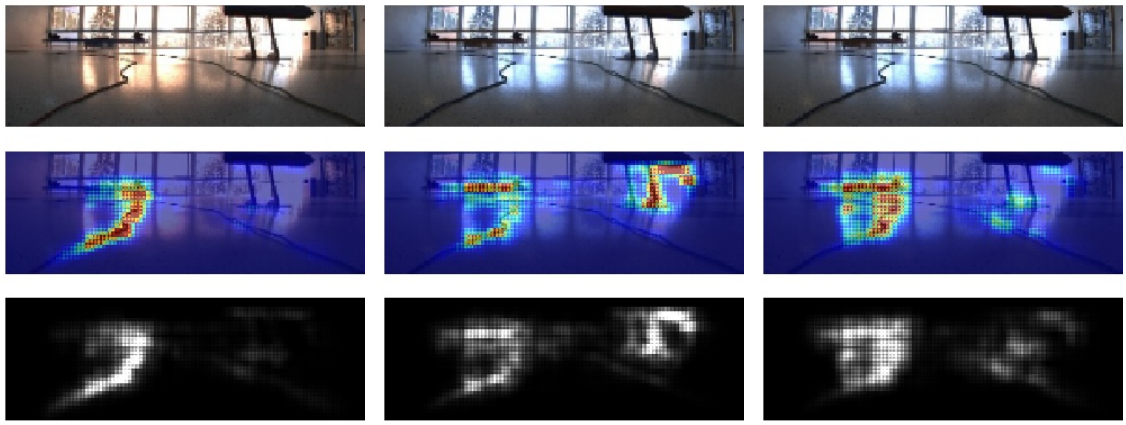
As seen in Figure 4.24, applying the horizontal shifts to the training data causes the validation loss to increase. However, testing the $SpurvPilot_{Udacity}$ model inside the Udacity Simulator showed the opposite effect. The model was not able to finish the track if the horizontal shifts were excluded, or if the selection rate or intensity range



(a) Random erasing and HSV adjustment

(b) HSV adjustment

(c) None



(d) Random erasing and HSV adjustment

(e) HSV adjustment

(f) None

Figure 4.23: VBP heat maps for the same frames in the test set. The models are trained with different augmentation techniques. The leftmost images are from the final *SpurvPilot* model, the middle images are from the *SpurvPilot* model trained without random erasing, and the right images are from the *SpurvPilot* model trained without random erasing and HSV adjusment.

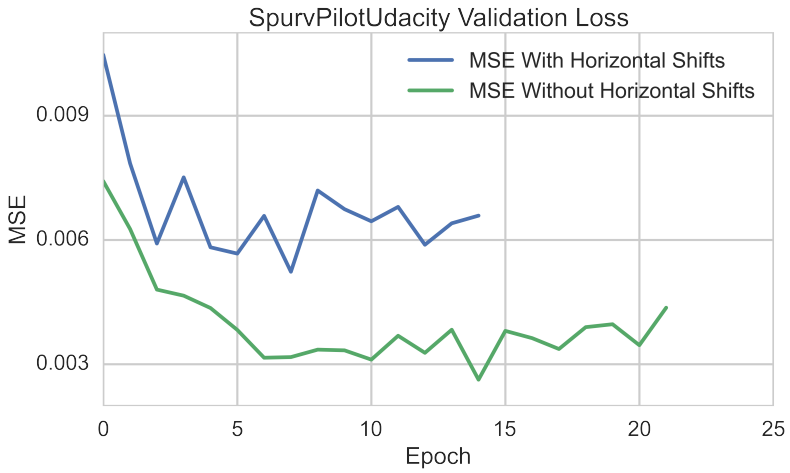
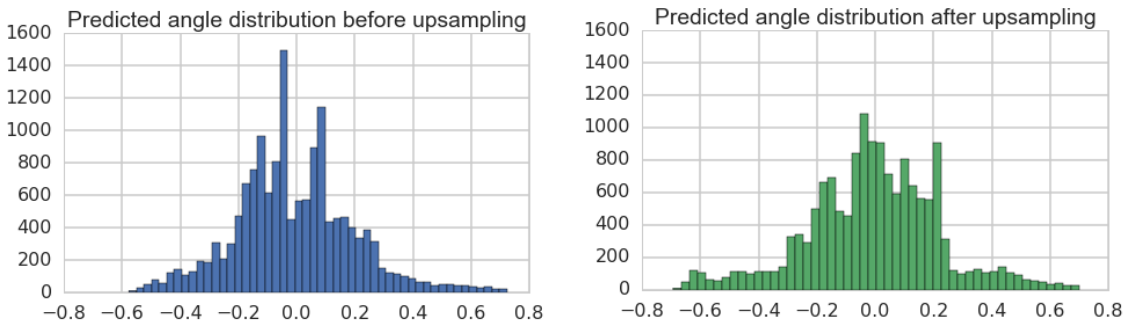


Figure 4.24: The validation loss of the $SpurvPilot_{Udacity}$ trained with and without the horizontal shifts augmentation technique.

was too low. The same was to some extent true for the usage of side camera images, but the model appeared to be less sensitive to those changes.

4.4.4 Upsampling

Figure 4.25(a) shows a histogram of the predicted angles by the *SpurvPilot* model on the test set, where the model is trained without upsampling of the angles in the training set as described in Section 3.5.1. How the predicted angle distribution is affected by upsampling is shown in Figure 4.25(b).



(a) Before upsampling **(b)** After upsampling

Figure 4.25: Histograms showing the distribution of predicted angles on the test set before and after the upsampling of angles larger than 0.5 radians by a factor of 5 in the training set.

4.4.5 Activation Functions

A histogram of predicted angles by *SpatialSpurvPilot* with and without using the Tanh activation function in the architecture's output layer is shown in Figure 4.26.

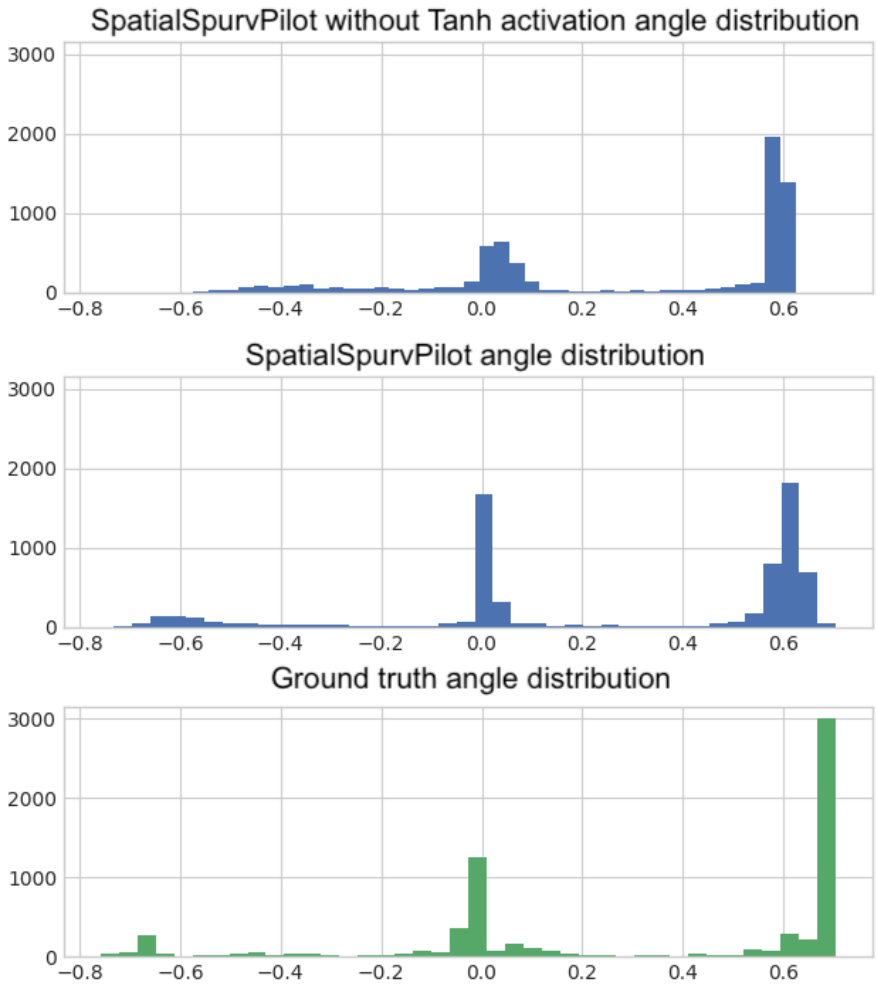


Figure 4.26: Comparison of *SpatialSpurvPilot* angle distributions with and without the output layer using the Tanh activation function instead of the linear function $f(x) = x$, showing how the Tanh activation affects the predicted angles.

Chapter 5

Discussion

In this chapter, the results from Chapter 4 are discussed and summarised. First, the results from the SPURV lane following task are considered in Section 5.1, followed by the Udacity simulator lane following task in Section 5.2. Discussion of the results of the marker turning task is done in Section 5.3, before Section 5.4 summarises advantages and problems with the deep learning framework used in this thesis. Finally, a summary of how the SPURV has performed as a tool for deep learning research is done in Section 5.5.

5.1 SPURV Lane Following

5.1.1 Performance and Generality

Quantitative Metrics

The quantitative metrics in Table 4.1 show that the error measures were generally high, with a MAE of ≈ 0.1 on the test set, which corresponds to an average deviation of 5.7 degrees from the ground truth. The quantitative measures discussed throughout this section do, however, need to be viewed critically, as explained in Section 5.4.1. For the SPURV lane following task, a noisy dataset can further contribute to the error. As mentioned in Section 3.5.1, steering the SPURV with the joystick could at times be tedious, which caused the collected data to contain inaccurate steering commands. The noise can be partly responsible for the high MSE and MAE values. Another source of inaccuracy in the dataset is the fact that the driver did not see the output from the camera during data collection, as mentioned in Section 3.2.1. This makes it very likely that situations occurred where the driver subconsciously performed a driving action based on information that was outside the field of view

of the camera. If such actions were present in the dataset, it would naturally make the task harder to learn while also resulting in high error rates.

The training error is higher than the validation error. This is due to the usage of generalisation techniques such as dropout and augmentation, which complicated the prediction task during the training phase.

Another notable aspect of the results is the significant difference between the validation and test error, which could suggest that the hyperparameters have been tuned too tightly to the minimisation of the validation loss, resulting in a model that is overfitted to the validation set. However, the gap may also be the result of an inadequately chosen test set, an issue that is discussed in Section 5.4.1.

The whiteness values in Table 4.1 and illustrations in Figure 4.2 show that the *SpurvPilot* generates more stable steering commands than the collected ground truth. The idea behind using RNNs for end-to-end steering angle prediction is that the predictions made by feed-forward neural networks can be unstable between subsequent frames, causing a high whiteness level. This was not the case for the indoor lane following task, possibly because the task is done in surroundings that are visually too simple for these issues to occur. Outdoors driving would naturally happen in more challenging visual surroundings, that could make an end-to-end steering angle predictor more unstable than in simple indoors surroundings. This may be reflected by the $SpurvPilot_{Udacity}$ model obtaining a higher whiteness value. It should be noted that the whiteness does not measure long-term instability, for instance, slowly swaying from side to side.

VBP Heat Maps

The selection of VBP heat maps in Figure 4.4 and the internal videos presented in Section 4.1.4 show that the *SpurvPilot* mostly focuses on the ropes. This was an important achievement, as the principle behind end-to-end learning is that the CNN can implicitly learn what the most relevant features in the image are. The network appears to have a tendency towards focusing on only one of the ropes, as in Figure 4.4(c), but an explanation for this has not been found.

Real-life Testing

As seen in the videos from the final real-life testing presented in Section 4.1.4, the *SpurvPilot* was able to successfully follow lanes on two tracks in two different locations. The layout of the test track is deemed sufficiently varied to confirm that the *SpurvPilot* has achieved the intended behaviour. The additional edge cases showed that the model was further able to handle even steeper turns than the ones in the test set, and also straighten up when in an erroneous position.

When considering the VBP heat maps, the performance on the track in *Location B* appeared to be better than in *Location D*. The heat map in Figure 4.4(e) suggests that the *SpurvPilot* is somewhat sensitive to dark, horizontal lines in the room, like moulding along walls. Such lines were more prominent in *Location D* than the other locations. The video "Real-life test of SpurvPilot model in "Location D" (Internal camera)" affirms that the *SpurvPilot* is distracted by the moulding. The distribution of training data in Table 3.1 shows that a majority of data points were from *Location B*, a spacious room with relatively few prominent horizontal lines. The minority of training data that was collected in *Location A*, which is a small room like *Location D*, did not contain as many visible lines as *Location D*. Adding some training data from smaller rooms with prominent lines would probably help cope with this problem. In conclusion, it looks like the *SpurvPilot* is not robust towards location changes.

Although care was taken to lay out varied tracks during data collection, it was to some extent inevitable that the nature of the rooms influenced the shape of the tracks. This caused the trained models to overfit to visually unique objects in the rooms, like the benches in *Location B*. This overfitting is visible in Figure 4.4(e) and Figure 4.4(f). Even more care should have been taken to vary the location and shapes of the tracks - although this is sometimes difficult in indoor locations where the amount of space is limited. The limited possibility of variation can be viewed as a general drawback of indoor training.

Prediction Time

The *SpurvPilot* model exhibited stable behaviour during the real-life testing with a camera frame rate of 15 fps. In the context of indoor lane following without moving obstacles, this appears to be more than sufficient. The maximum detection speed of 62 fps is fairly high compared to other end-to-end learning efforts in autonomous vehicles, like PilotNet's 30 fps, but is difficult to compare because of the usage of different computing libraries, programming languages and GPUs. No attempts at optimising the model for fast predictions have been made and could be a natural extension of this thesis, especially in the context of obstacle avoidance or speed control. The already high prediction speed does indicate that the SPURV is capable of running more complicated ANN architectures without reaching unacceptable prediction times.

5.1.2 Relevance

The SPURV lane following environment is a very simplified version of a real road environment. Most notably, the appearance of real roads can vary greatly and the road outlines are usually not as visible as ropes on an even surface. Still, the experiments were able to give insight into many important aspects of lane following, and end-to-end learning in general. One insight is the importance of

augmentation both to increase the generality of the model and to introduce it to edge case situations. Another important lesson learned is the need to investigate early on the relationship between quantitative regression metrics like MSE and qualitative results, like the VBP heat maps and real-life performance. The field of end-to-end learning for autonomous vehicles was found to lack a common practice for verifying and comparing performance.

5.2 Udacity Simulator Lane Following

5.2.1 Performance and Generality

The quantitative metrics in Table 4.2 show that both the MSE and MAE were lower for the $SpurvPilot_{Udacity}$ than for the $SpurvPilot$, while the whiteness was higher. The low MSE was not reflected in improved performance inside the simulation, compared to the $SpurvPilot$'s real-life performance. The low error might be due to the Udacity Simulator test set containing fewer and smaller turns than the SPURV lane following test set, which can be seen in Figure 3.18 and Figure 3.17. The high whiteness suggests that the performance of the $SpurvPilot_{Udacity}$ model is more unstable than $SpurvPilot$, which can be due to a more visually complicated environment. It can also be connected to the high ground truth whiteness of the Udacity Simulator data, which suggests that the model had to learn from more fluctuating data. Still, Figure 4.9 shows that the $SpurvPilot_{Udacity}$ performed better than the ground truth. From the quantitative metrics alone, it is again difficult to draw conclusions about the performance of the model and to compare the performance of $SpurvPilot_{Udacity}$ to the $SpurvPilot$.

Simulation Tests

The model was able to drive autonomously on the Lake Track, as shown in the video "Simulator test of $SpurvPilot_{Udacity}$ on the Lake Track". However, the driving behaviour was not perfect. The car swung from side to side several places and did almost exit the lane once. The errors seemed to appear at locations containing prominent shifts in the appearance of the track, for instance when approaching and leaving the bridge, which looks very different from the remaining parts of the track.

Concerning generality, there is a risk that the model is overfitted to the Lake Track, since it is both trained and tested on data from that exact track. If this is the case, the model relies on recognising the location of the track, rather than the outline of the roads. The VBP heat maps in Figure 4.10 show that although the network does focus on the outlines of the road, other features of the image receive a considerable amount of attention. In the testing video, the $SpurvPilot_{Udacity}$ is close to exiting

the lane but is able to straighten itself up. This suggests that the $SpurvPilot_{Udacity}$ at least does not solely rely on overfitting, since the training data does not contain these out-of-the-ordinary situations. Testing on a separate track with a similar road layout would have been ideal but was not available. Without this option, it is not possible to conclude about the overfitting of the $SpurvPilot_{Udacity}$ model.

Pretraining

From inspection of the VBP heat maps, it was found that the model did not focus as much on the outlines of the roads as hoped for. Since the *SpurvPilot* model trained for doing the SPURV lane following task focused heavily on the ropes, i.e. the lane "outlines", its weights were used to initialise the $SpurvPilot_{Udacity}$ model. This pretraining appeared to have the desired effect, where the model focused more on the lanes in the Udacity simulator than it did without pretraining, as seen in Figure 4.22.

5.3 Marker Turning

5.3.1 Performance and Generality

Quantitative Metrics

The quantitative metric results in Table 4.3 show that the MSE on the training set is much smaller than on the validation set for both the *SpatialSpurvPilot* and *SpatialSpurvPilotExtended* models, even though both models were trained with dropout in all hidden layers, which increases the training error. This means that both models perform better on the samples they have been trained on than unseen samples, which is a sign of overfitting. The high validation MSE can, once again, be connected to low-quality validation data, but nevertheless suggests that the *SpatialSpurvPilot*'s performance is unsatisfactory.

The marker turning dataset contains some amount of noise. Although care was taken during data collection to keep the behaviour consistent, it was inevitable that the driver behaviour had small variations. The distance from the cone at which the turns were initiated was especially tricky to keep consistent.

Another negative property of the training set is that around 1% of it is guaranteed to be junk data. The junk data comes from the changes of locations, as seen in Table 3.3, which were not considered when combining frames for spatial history. The change between *Location B* and *Location D* resulted in $2 \times D_f = 160$ corrupt spatial history frames, so did the change between the takes with and without markers. This

1% of junk data was so small in the context of the entire training set that it was considered negligible, but in retrospect, it might have contributed to the instability of the model.

Path Visualisation

The path visualisations were particularly useful for this task, as they are a quick and simple technique for checking whether or not a trained model seems to be able to solve the task. Already by looking at the path visualisations generated for the validation set in Figure 4.15, one could see indications of the issues that would later be uncovered during real-life testing, as summarised in Figure 4.19.

A possible extension of the path visualisation could be to shift the image with some factor of the difference between the ground truth path and the predicted path and input this image into the model for the next frame. This would be equivalent to an interactive simulator as described in Section 2.2.3. As testing was easily available on the SPURV, this was not prioritised for this project but could be an interesting improvement in future research - especially if the domain is real outdoor roads, where safety and availability are more of a concern.

Activation Heat Maps

Indications of overfitting could be spotted in the generated VBP heat maps. While *SpatialSpurvPilot* seems to focus almost exclusively on the markers in the validation set, as in Figure 4.16, *SpatialSpurvPilotExtended* appears to also focus on surrounding objects, as shown in Figure 4.17. This could indicate that the *SpatialSpurvPilotExtended* model is more overfitted than *SpatialSpurvPilot*. This makes sense, as the former has more parameters to tune due to its additional fully connected layer while being provided with the same amount of training data as *SpatialSpurvPilot*.

SpatialSpurvPilotExtended was designed due to the concern that the largest fully connected layer of *SpatialSpurvPilot* would not be able to capture the entire feature map from the convolutional layers. Because of the simple indoor environment, this extension seems to have been completely unnecessary, as *SpatialSpurvPilot* performed just as well as *SpatialSpurvPilotExtended*, and with less detected overfitting.

Real-life Testing

Real life testing uncovered unstable behaviour in both models. A large portion of the failures during this testing seemed to be connected to the turning time issues described in Section 4.3.4, which again caused the spatial history to lose track of the markers.

Turning up the D_f parameter slightly in the spatial history during real-life testing could maybe help the current *SpatialSpurvPilot* and *SpatialSpurvPilotExtended* models complete their tasks more stably, but would not solve the actual issue: That the models have failed to learn the optimal behaviour for solving the marker turning task.

5.3.2 Training and Spatial History

The usage of spatial history introduced some problems with utilising the deep learning framework applied to the single input image approach used in the lane following tasks. Namely, a majority of the augmentation techniques mentioned in Section 3.3.3 could no longer be used. The upsampling of large steering angles could also not be applied, as the training data had to be coherent to be correct. These issues will now be discussed in detail.

Augmentation

The majority of augmentation techniques applied to the SPURV and Udacity simulator lane following tasks were ruled out as augmentation techniques for the marker turning task due to the risk of corrupting the training data. Horizontal flipping would only confuse the model, as right and left turns were to be done in specific situations. Due to the nature of the marker turning task, horizontal shifts on the most recent image would not generate valid training data. Random erasing ran the risk of accidentally erasing a highly relevant marker from the spatial history. This left only HSV adjustment as an applicable augmentation technique. This lack of augmentation could partly explain the poor performance, especially in heavy backlight, during real-life testing.

Activation Function

As seen in Figure 4.26, the architecture version with linear activation instead of Tanh activation in the output layer struggled to predict the largest angles, as was also the case in the SPURV lane following task. In the latter, this problem was quickly fixed by upsampling the largest angles in the training set, by simply copying these samples by a factor of 5. This was no longer a valid solution when spatial history was introduced, as the training data had to be coherent in order to be valid, and such upsampling would not ensure coherent data. Instead, changing the activation function to the nonlinear Tanh function was a cleaner approach that had the desired effect.

5.3.3 Relevance

In the context of the works by Hubschneider et al. (2017), who were able to make a model respond to navigational inputs, the marker turning task can be considered a more complicated extension in a simplified environment. It is more complicated because, unlike Hubschneider et al. (2017), there is no input signal indicating that a certain action is supposed to be taken, and the model is forced to recognise the right action based exclusively on visual cues. This situation is close to real traffic environments, where certain visual cues need to initiate certain actions, like responding to road work and blockages, or turning around when reaching the end of a road. The environment of the marker turning task is simplified, because the model does not have to simultaneously perform a difficult additional task, like lane following. In the task of Hubschneider et al. (2017), the model had to execute the turning while still following the road correctly, incorporating rather complex decision making. A natural extension of the marker turning task was, therefore, to have the model perform the U-turns while also forcing it to be more aware of its surroundings. However, since the model was not able to adequately perform the original task, no extensions were attempted.

While not being directly tied to a specific task for autonomous vehicles, the marker turning has given insight into how a spatial history works in a CNN, and its limitations. The main limitations with spatial history were identified as limited data augmentation possibilities and the risk that important information gets lost between the frames in the history. The longer prediction time is another disadvantage.

5.4 Deep Learning Framework

5.4.1 Quality of Validation and Test Loss

The correctness of the final quantitative metrics presented in Section 4.1.1 may suffer from a validation and test set that was not functioning as expected. The validation loss was a challenge to work with for both of the lane following tasks. During the first attempts at training, as mentioned in Section 3.3.2, the validation loss was suspiciously small, as the set of randomly chosen images in the validation set had almost identical sister points in the training set. Care was then taken to make sure that the validation set consisted of completely separate tracks, to make sure the models were not overfitted to the tracks. A persistent problem henceforth was that the validation loss behaved strangely, often not decreasing at all, but staying near constant or even increasing. Experimentation with different optimisers and learning rates was attempted, and far along helped battle this problem. However, when examining the VBP heat maps, the car path visualisations and the performance during real-life testing, a clear relation between a good validation loss and real-life

performance was not found. It was uncovered, both for the SPURV lane following task and the Udacity task, that changing the validation set dramatically affected both the behaviour of the validation loss and the final results, which means that the variance of the validation set was high.

The validation and test set issues might be the result of a poorly chosen validation set. In retrospect, a larger and more diverse validation set should have been chosen for the lane following task. Preferably, at least one track from each location should have been selected. Extra care should have been taken when recording the validation and test set to make sure that the behaviour of the car was as accurate and consistent as possible. The variance and quality of the validation set performance should have been explored early on, to allow more time to perform the actions above. K-fold cross-validation would have helped battle the validation set variance but was not used due to the considerable increase in training duration. Since the test set was chosen in the same manner as the validation set, it is natural to assume that these issues affected the test set as well.

From this, one can conclude that choosing good validation and test sets in end-to-end learning for autonomous vehicles is a difficult problem. Collecting adequate data for autonomous vehicles that are supposed to operate in real-life environments, is a highly important and tough task. Verifying that these sets of data cover all possible edge cases is virtually impossible. Even for simple indoor tasks as the ones in this thesis, the choice of testing and validation sets was an intricate issue. Hence, the empirical results on these data collections cannot be regarded alone when evaluating end-to-end autonomous vehicle systems.

Quality of Quantitative Metrics

The issues discussed above can also to some extent be explained by the general issues regarding the evaluation of end-to-end self-driving cars, as presented in Section 2.2.3. Another notable aspect of the validation and test losses is that they cannot be used to assess the effect of some of the measures taken to increase the robustness of the model, like applying horizontal shifts to the training set. Figure 4.24 shows that this augmentation technique had a negative effect on the validation loss, even though the model performed better, as described in Section 4.4.3. This is due to the validation set not containing any examples of the unfamiliar situations simulated by the horizontal shifts. More experiments should have been made to gain more insight into the validation loss's ability to reflect real-life performance.

Due to the instability of the validation loss, the VBP heat maps and car path visualisations were used extensively in this thesis to determine hyperparameters and compare architectures. More quantifiable evaluation methods would have been beneficial both for the quality and progress of the thesis.

5.4.2 Architecture

The *SpurvPilot*, *SpatialSpurvPilot* and *SpatialSpurvPilotExtended* architectures were all based on other work where vehicles predicted angles in real outside traffic situations. Hence, it is very likely that these networks are much larger than necessary for the indoor tasks in this thesis, as the surroundings the SPURV operated in, were much less visually challenging than in an outside environment.

5.4.3 Hyperparameters

The hyperparameter combination that was found when training the *SpurvPilot* model, was used without major changes in the Udacity simulator lane following task as well as the marker turning task without further tuning. It is, therefore, possible that there might exist better hyperparameter configurations for the two latter tasks.

The hyperparameter tuning could be streamlined by using a technique called grid search, where all permutations of possible hyperparameters are automatically tested and selected to optimise some selection of metrics. Unfortunately, this was difficult to apply to the particular problems in this thesis, as manual inspection of the VBP heat maps were an essential part of the evaluation.

5.4.4 Augmentation

Figure 4.23 shows examples of how the applied HSV adjustments and the random erasing made the model focus less on windows and light reflections and focus more on the ropes. This suggests that these augmentation techniques can be powerful tools to help models handle images with difficult light conditions, which is described as an issue by several articles on end-to-end autonomous vehicles. HSV adjustments, or similar image adjustments, are well-known augmentation techniques within the field, but examples of using random erasing were not found. It could be interesting to experiment more with this augmentation technique in outdoor environments.

All in all, augmentation has been established as a crucial tool for improving the generality and robustness of end-to-end lane following systems.

5.5 The SPURV as a Research Tool

5.5.1 The SPURV Setup

The SPURV Research had just been acquired by NTNU when the work on this thesis started. The SPURV Research from KVS is a fairly recently released product, intended to be used as a general framework for academic experimentation. The documentation was thus limited and had some incomplete parts, especially for people who are unfamiliar with the used technology stack, like the team behind this thesis. Hence, the setup, configuration and development needed to start collecting usable data from the SPURV took a considerable amount of time. For instance, apart from how to record a bag, every aspect of the data gathering, including I/O and storage issues, synchronisation issues and the conversion from the bag format, had to be figured out without any help from the documentation. The crew at KVS Technologies was very helpful during this process, answering questions quickly and even adding new guides to the documentation when requested. The SPURV documentation has since the start of this project been continuously expanded. The efforts made throughout this thesis that can be relevant to other SPURV Resarch users have been compiled in a collection of user guides in Section A.2.

5.5.2 Using the SPRUV for Data Collection

Once set up, the final data processing pipeline, including the synchronising, transferring, converting, cleaning and reducing data, worked smoothly and relatively efficiently. One issue, however, was the massive time delay of the video stream from the SPURV to the host laptop, which made steering using solely the camera view from the SPURV impossible. Hence, steering was done from the external viewpoint of a human, which means that information hidden from the camera was taken into account in the decision making. For the same reason, the laptop needed to be carried while walking behind the SPURV and simultaneously steer it with the Xbox controller, which was slightly impractical. According to KVS Technologies, the delay issue can be solved by rewriting the Joystick Driver-node to use gstreamer[1], but this was not prioritised.

A big advantage of using the SPURV to gather data for autonomous driving systems, especially if compared to a real, life-sized car, is that it is simple, affordable and very safe, while not requiring any expensive resources or extensive planning. Moreover, the SPURV can be tested for situations that it is hard to put real cars in. A disadvantage of using the SPURV is the lack of realism in the test environment, as it cannot be put in direct vehicular traffic.

[1] `https://gstreamer.freedesktop.org/`

5.5.3 Using the SPURV for Testing Neural Networks

The SPURV Research vehicles had not previously been used for actively testing a deep learning model, but KVS Technologies had tested that it was possible to run an ANN on its on-board computer. The technical specifications of the NVIDIA Jetson TX2 GPU make the SPURV capable of running reasonably large neural networks with good prediction times. This project work has brought to attention that the ability to perform interactive, real-life tests is a crucial part of developing end-to-end autonomous vehicles.

Putting the SPURV in situations that are entirely equivalent to real-life vehicles is practically not possible. However, it can be utilised as a useful tool for initial testing of new ideas or network architectures in limited environments to see if they have any potential to work, before starting a comprehensive data collection and testing process using real vehicles in real-life traffic.

Chapter 6

Conclusion and Future Work

In this chapter, a conclusion based on the results in Chapter 4 and the following discussion in Chapter 5 is formulated. Then, pointers to interesting directions in future research are presented.

6.1 Conclusion

In this thesis a feed-forward neural network has been trained through an end-to-end approach to predict steering angles for an indoors lane following task using the SPURV vehicle. The trained model was evaluated through a combination of quantitative and qualitative evaluation metrics. It was found that few quantitative metrics were able to give clear answers about the actual performance of the system. A metric that measured the difference between the predictions in consecutive frames, whiteness, provided useful information about the stability of the steering angle predictor. Additionally, visualisations of the path of the car given the predicted steering angles along with heat maps obtained through the VBP technique, provided useful insight into the performance and generality of the trained model. During real-life testing on the SPURV vehicle, the final model was able to follow two indoor tracks in different locations without problems, while also correcting its position when it was put in erroneous initial orientations. Findings show that the choice of augmentation techniques had a severe impact on the performance of the models. The random erasing technique seemed to help the model handle image occlusion due to difficult lighting conditions, while horizontal shifting of the images helped the model learn how to straighten up if put in an erroneous position. As the approach used to train and evaluate this end-to-end system gave good results, the first research question, **RQ1**, has been answered.

RQ2 asked if it was possible to teach an end-to-end feed-forward neural network to

perform a complex manoeuvre without any other training information than visual queues and steering commands. To investigate this, a spatial history containing three images with a constant spatial position difference of the car were sent as input to the network. The results from the real-life testing showed unstable behaviour, where the models sometimes were able to perform the U-turning task, while sometimes failing completely. Nevertheless, the VBP heat maps and the positive parts of the real-life testing results indicate that a feed-forward neural network can learn such a complex manoeuvre if provided with enough high-quality training data and enough time spent on hyperparameter tuning. Thus, the second research question is partially answered.

6.2 Future Work

This section provides pointers to interesting approaches in future research, while also providing some suggestions of how to improve the SPURV as a data collection tool.

6.2.1 SPURV Data Collection Improvements

A necessary improvement to the data collection setup on the SPURV is optimisation of the camera image transfer speed between the SPURV robot and the host laptop. With this improvement, the driver could watch a live stream of images from the SPURV, and perform their actions exclusively on the same information as a model would have access to. This would result in better quality training, validation and test data.

6.2.2 Simulator Tool

As described in Section 2.2.3, it appears to be a tendency that researchers and companies spend a lot of resources to implement their own tools for evaluating their autonomous vehicle systems, using a wide variety of often non-comparable metrics. This makes it difficult to compare one research effort to another. A suggestion is using the path visualisation approach in this thesis and expanding it to create an interactive autonomous vehicle simulator. An evaluation system with a wide range of well-defined metrics and an API for inputting test data could introduce a unified way of evaluating autonomous vehicle systems or at least help researchers save time and resources by sparing them from implementing their own evaluation tools with small variations.

6.2.3 Architectures

This section proposes some ideas for experimenting with various ANN architectures and approaches.

Adapting Network Sizes

The ANN architectures used in this thesis are large, especially considering the indoors environment. Minimising the architectures to achieve optimal prediction times and minimise the number of parameters could be useful, especially in dynamic environments.

Recurrent Neural Networks

The spatial history approach used in the marker turning task could be compared to a RNN, like LSTM, as both approaches can be seen as serving the same function when used for end-to-end autonomous driving. Determining which of the architectures that has the best performance, and any limitations of either would arguably give a useful contribution to the end-to-end autonomous vehicle field.

Proxy Task

An interesting extension of the marker turning task could be to give the model the proxy task of identifying that it is doing a U-turn. This could be done by adding an output node, which the model should activate whenever in a turn. Some manual labelling of the training set would be required but could improve the performance on the marker turning task as one would have the guarantee that the model distinguishes the act of performing a U-turn manoeuvre from regular driving. This could also be extended to other manoeuvres that would be necessary during driving, like turning at intersections or roundabouts.

6.2.4 Tasks

As the SPURV is very suitable for data collection and real-life experimentation, it can be used to experiment with a variety of autonomous driving tasks. A few are suggested below.

Speed Control

A natural extension of the tasks in this thesis is to perform end-to-end throttle and brake control prediction in addition to predicting steering angles. This could, for instance, include static or dynamic obstacles in the SPURV lane following task.

Using the Back Camera

The speed control extension above could be further expanded by utilising the backwards facing camera on the SPURV. The data from this camera could be used to learn complicated manoeuvres like turning around in narrow spaces where backing up would be a necessary part of the manoeuvre.

Going Outdoors

As the SPURV is equally well equipped for driving outdoors as indoors, it would be interesting to for instance perform sidewalk following or a similar task outdoors. It could also be interesting to expand the indoor lane following dataset by setting up a lane following track outside. This would diversify the surroundings in which the SPURV would be able to do lane following in, something that might be easier to do outdoors than indoors. Due to its small size, it would not be suitable to put the SPURV in real car traffic.

References

Abadi, Martín et al. (2016). "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pp. 265–284. ISSN: 0270-6474. arXiv: 1605.08695. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Bojarski, Mariusz, Anna Choromanska, et al. (2016). "VisualBackProp: efficient visualization of CNNs". In: arXiv: 1611.05418. URL: http://arxiv.org/abs/1611.05418.

Bojarski, Mariusz, Davide Del Testa, et al. (2016). "End to End Learning for Self-Driving Cars". In: pp. 1–9. arXiv: 1604.07316. URL: http://arxiv.org/abs/1604.07316.

Bojarski, Mariusz, Philip Yeres, et al. (2017). "Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car". In: *CoRR* abs/1704.0, pp. 1–8. arXiv: arXiv:1704.07911v1. URL: http://arxiv.org/abs/1704.07911.

Caltagirone, Luca et al. (2017). "LIDAR-based Driving Path Generation Using Fully Convolutional Neural Networks". In: *arXiv*. arXiv: 1703.08987. URL: http://arxiv.org/abs/1703.08987.

Cermak, Jiri and Anelia Angelova (2017). "Learning with proxy supervision for end-to-end visual learning". In: *IEEE Intelligent Vehicles Symposium, Proceedings*(Iv), pp. 1–6. DOI: 10.1109/IVS.2017.7995690. URL: https://research.google.com/pubs/pub45985.html.

Chan, Ching-Yao (2017). "Advancements, prospects, and impacts of automated driving systems". In: *International Journal of Transportation Science and Technology*. ISSN: 20460430. DOI: 10.1016/j.ijtst.2017.07.008. URL: http://linkinghub.elsevier.com/retrieve/pii/S2046043017300035.

Chellapilla, Kumar, Sidd Puri, and Patrice Simard (2006). "High Performance Convolutional Neural Networks for Document Processing". In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. URL: https://hal.inria.fr/inria-00112631.

Chen, Chenyi et al. (2015). "DeepDriving: Learning affordance for direct perception in autonomous driving". In: *Proceedings of the IEEE International Conference*

on Computer Vision. Vol. 2015 Inter, pp. 2722–2730. ISBN: 9781467383912. DOI: 10.1109/ICCV.2015.312. arXiv: 1505.00256.

Chen, Zhilu and Xinming Huang (2017). "End-To-end learning for lane keeping of self-driving cars". In: *IEEE Intelligent Vehicles Symposium, Proceedings*, pp. 1856–1860. ISBN: 9781509048045. DOI: 10.1109/IVS.2017.7995975.

Chetlur, Sharan et al. (2014). "cuDNN : Efficient Primitives for Deep Learning". In: *CoRR* abs/1410.0, pp. 1–9. arXiv: arXiv:1410.0759v3. URL: http://arxiv.org/abs/1410.0759.

Chollet, Francois (2017). *Deep learning with Python*. Manning Publications Co. ISBN: 9781617294433.

Dickmanns, E.D. et al. (1994). "The seeing passenger car 'VaMoRs-P'". In: *Proceedings of the Intelligent Vehicles '94 Symposium*, pp. 68–73. DOI: 10.1109/IVS.1994.639472. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=639472.

Eraqi, Hesham M., Mohamed N. Moustafa, and Jens Honer (2017). "End-to-End Deep Learning for Steering Autonomous Vehicles Considering Temporal Dependencies". In: (Nips), pp. 1–8. arXiv: 1710.03804. URL: http://arxiv.org/abs/1710.03804.

Geiger, A. et al. (2013). "Vision meets robotics: The KITTI dataset". In: *International Journal of Robotics Research* 32(11), pp. 1231–1237. ISSN: 02783649. DOI: 10.1177/0278364913491297. arXiv: 1102.0183.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press: Cambridge, MA. ISBN: 9780262035613. URL: http://www.deeplearningbook.org.

Hubschneider, Christian et al. (2017). "Adding Navigation to the Equation: Turning Decisions for End-to-End Vehicle Control". In: *IEEE International Conference on Intelligent Transportation*(October). DOI: 10.1109/ITSC.2017.8317923.

LeCun, Yann et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86(11), pp. 2278–2323. ISSN: 00189219. DOI: 10.1109/5.726791. arXiv: 1102.0183.

LeCun, Y et al. (2006). "Off-road obstacle avoidance through end-to-end learning". In: *Advances in neural information processing systems* 18, p. 739. ISSN: 1049-5258. URL: http://papers.nips.cc/paper/2847-off-road-obstacle-avoidance-through-end-to-end-learning.

Martinez, Mark et al. (2017). "Beyond Grand Theft Auto V for Training, Testing and Enhancing Deep Learning in Self Driving Cars". In: pp. 1–15. arXiv: 1712.01397. URL: http://arxiv.org/abs/1712.01397.

Pomerleau, Dean (1989). "Alvinn: An autonomous land vehicle in a neural network". In: *Advances in Neural Information Processing Systems 1*, pp. 305–313.

Quigley, Morgan, Brian Gerkey, and William D Smart (2015). *Programming Robots with ROS: a practical introduction to the Robot Operating System*. First Edit. O'Reilly Media, Inc.

ROS Wiki (n.d.). URL: http://wiki.ros.org.

Russakovsky, Olga et al. (2014). "ImageNet Large Scale Visual Recognition Challenge". In: *CoRR* abs/1409.0575. arXiv: 1409.0575. URL: http://arxiv.org/abs/1409.0575.

Russel, Stuart and Peter Norvig (2014). *Artificial Intelligence - A Modern Approach*. Pearson Education Limited. ISBN: 9781292024202.

Sallab, Ahmad El et al. (2016). "End-to-End Deep Reinforcement Learning for Lane Keeping Assist". In: (Nips), pp. 1–9. arXiv: 1612.04340. URL: http://arxiv.org/abs/1612.04340.

Santana, Eder and George Hotz (2016). "Learning a Driving Simulator". In: *CoRR* abs/1608.01230. arXiv: 1608.01230. URL: http://arxiv.org/abs/1608.01230.

Simonyan, Karen, Andrea Vedaldi, and Andrew Zisserman (2013). "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps". In: pp. 1–8. arXiv: 1312.6034. URL: http://arxiv.org/abs/1312.6034.

Springenberg, Jost Tobias et al. (2014). "Striving for Simplicity: The All Convolutional Net". In: pp. 1–14. ISSN: 02548704. arXiv: 1412.6806. URL: http://arxiv.org/abs/1412.6806.

Yang, Zhengyuan et al. (2018). "End-to-end Multi-Modal Multi-Task Vehicle Control for Self-Driving Cars with Visual Perception". In: arXiv: 1801.06734. URL: http://arxiv.org/abs/1801.06734.

Yosinski, Jason et al. (2014). "How transferable are features in deep neural networks?" In: pp. 1–9. arXiv: 1411.1792. URL: http://arxiv.org/abs/1411.1792.

Zhong, Zhun et al. (2017). "Random Erasing Data Augmentation". In: arXiv: 1708.04896. URL: http://arxiv.org/abs/1708.04896.

Appendices

Appendix A

SPURV

A.1 Specification

Dimensions	500 x 450 x 250 mm without antennas
Weight	9 kg
Max speed	45 km/h (12 m/s)
Operation time	3 hours with normal use
Battery charging voltage	100 - 240V AC
GPU	NVIDIA Tegra X2
Front camera	FLiR Machine Vision – Blackfly S 16S2C
Back camera	OnSemi AR0330 3.4MP USB3 1080p60

Table A.1: Hardware specification of SPURV car.

Operating system	Ubuntu 16.04 (Xenial) r28.1
NVIDIA Jetpack version	3.1
CUDA version	8.0
ROS version	ROS Kinetic Kame
TensorFlow	1.6.0
Keras	2.1.5

Table A.2: Software specification of SPURV car.

A.2 User Guides From This Thesis

This section contains comprehensive guides for all SPURV-related setup and tasks related to this thesis. The goal is to simplify the process of using the SPURV for research purposes in future projects. This section is written with the assumption that the host machine runs on Ubuntu 16.04.

A.2.1 Connecting to the SPURV

We separate between the host machine and the SPURV itself. The host machine is typically a laptop, where for instance control (like an XBOX-controller) and image viewing nodes run. When the SPURV is turned on, one can connect to it by connecting to the ResearchSPURV WiFi-network.

One can further gain access to the files on the SPURV by connecting to it through SSH,

```
ssh nvidia@spurv
```

where `spurv` either is the keyword configured in your SSH config or the IP address of the SPURV directly.

Whenever we mentioned something being done "on the SPURV" in this guide, we mean by being connected to the SPURV through SSH.

A.2.2 ROS Installation on Host Machine

Installing ROS

Follow the instructions at `http://wiki.ros.org/kinetic/Installation/Ubuntu`

Installing Aravis

```
# Clone the Aravis repository to the location of your choice
git clone https://github.com/AravisProject/aravis.git

cd aravis

sudo apt-get install autoconf
```

```
sudo apt-get install intltool
sudo apt-get install gtk-doc-tools

./autogen.sh
./configure
make
make install
```

Building the SPURV Workspace on Host Machine

```
sudo apt-get install libgstreamer1.0-dev
sudo apt-get install libgstreamer-plugins-base1.0-dev

rosdep install --from-paths src --ignore-src -r -y
```

Replace the `src`-folder in the workspace you just created with the `src` folder copied from the SPURV.

Give all the files in `spurv_research/spurv_examples/src` execution permissions by running

```
chmod +x *.py
```

in the folder.

Then build the workspace:

```
catkin_make
```

Suggested Shell Setup

We recommend configuring your shell in the following way. We assume you have defined the aliases mentioned here in the rest of this user guide. Add the following to your `~/.bashrc` or equivalent file:

```
# installation
source /opt/ros/kinetic/setup.bash
source YourWorkspace/devel/setup.bash

# ip addresses
```

```
export ROS_MASTER_URI=http://192.168.1.5:11311/ # IP of SPURV
export ROS_IP=192.168.1.46 # Your IP, given by the SPURV's DHCP
    server

# aliases for changing ros master
# useful for steps in  Section A.2.4  and Section A.2.4
alias rosmasterlocal='export ROS_MASTER_URI=http://127.0.0.1:1234
    && export ROS_IP=127.0.0.1'
alias rosmasterspurv='export
    ROS_MASTER_URI=http://192.168.1.5:11311 && export
    ROS_IP=192.168.1.46'
alias roscorelocal='roscore -p 1234'

alias sync_rosbag='python 'Path/to/sync/script'
```

A.2.3 Configuration

Setting Camera Resolution

The resolutions of the cameras can be set in the following files:

```
spurv_research/spurv_launchers/launch/bwd_camera.launch
spurv_research/spurv_launchers/launch/fwd_camera.launch
```

Any other configuration files do not seem to have any effect. These files need to be changed **on the SPURV**, not on the host machine. The SPURV has to be restarted for the changes to take effect.

Follow the camera resolution guide in the KVS User Manual. Any resolution changes to `fwd_camera` will only crop the image in the upper left corner, and thus the resolution cannot be reduced. `bwd_camera` supports given combinations of resolution and framerate, see the KVS manual.

See Section A.2.4 for an explanation on how to add time-stamps to the messages sent from the camera node, if this is needed for your use case.

A.2.4 Data Collection

This is the guide for data collection. Make sure you read the *Synchronising rosbags* section before you actually start collecting if this is a brand new SPURV.

Joystick

If you have an XBOX-controller connected to the host machine, and it does not get listed when you run `ls /dev/input` as `jsX`, where `X` is some integer, you need to install some drivers and run them:

```
sudo apt-get install xboxdrv
sudo xboxdrv
```

Then, follow this guide:

`http://wiki.ros.org/joy/Tutorials/ConfiguringALinuxJoystick`

You'll have to start a joy node as described in the link above for the input from the joystick to be published to the `/joy` topic.

Collecting Data in Rosbags

We recommend to start the rosbags collections directly on the SPURV, and save the rosbag files on the SD card that is mounted there.

The mount point of the SD card can be found in `/media/nvidia/NTNUspurvSD`.

You can start a new recording to a rosbag file in the folder you are currently in by running

```
rosbag record <topics> -b 2048
```

Replace `<topics>` with a list of the topics you wish to record. You can list all available topics by running `rostopic list`.

The `-b` flag increases the buffer size. We recommend setting it to at least 2048 - that seemed to be enough for us.

If you want to record the front camera, we recommend getting the compressed images as it takes too much bandwidth saving the original ones.

Example: Saving front camera, back camera, and steering commands:

```
rosbag record /fwd_camera/image_raw/compressed
   /bwd_camera/image_raw/compressed /ackermann_cmd -b 2048
```

More on rosbags here: `http://wiki.ros.org/rosbag/Commandline`

Transfer data to laptop

Transfer to host machine over ssh. While on the host machine:

```
scp -r nvidia@spurv:/media/nvidia/NTNUspurvSD/<your rosbag folder>
   <your path locally>
```

Using `rqt`

You can use the tool `rqt` to review and play rosbags on the host machine. For this, you'll need to change the ROS master IP from the SPURV to the localhost on the host machine (as it comes in handly viewing these files while not always being connected to the SPURV).

In two different terminal windows, run:

```
rosmasterlocal
roscorelocal
```

```
rosmasterlocal
rqt
```

A window with the tool will then open, and you can view the different topic data by right-clicking it and choosing whether you wish to see the raw data, a graph or images.

Synchronising Rosbags

If nothing else is defined, messages saved in rosbags get a timestamp in the moment they are received, which often deviates from the moment they were created.

Originally, the images from the SPURV did not have a defined timestamp. To fix this, one can add the line `image.header.stamp = ros::Time::now()` right before the line `_imgPub.publish(image);`.

On the SPURV, make these changes in both

`spurv_reserach/spurv_camera/src/aravis_camera_node.cpp` and
`spurv_research/spurv_camera/srcV4lCameraNode.cpp`.

You'll have to run `catkin_make` on the SPURV and restart it for the changes to take effect.

By running

```
sync_rosbag <your_rosbag>
```

a new bag that is synchronised will be created, by replacing the main timestamp with the timestamp of when each message was created.

Converting Rosbags to CSV Files and Images

The file `collect_training_data` has been provided to us by Revolve NTNU. By playing a rosbag file while `collect_training_data` is running, the data being published by the bag will be converted to a CSV-file of steering commands along with images.

In three different terminal sessions, run:

```
rosmasterlocal
roscorelocal
```

```
rosmasterlocal
rosrun collect_training_data collect_training_data.py
```

```
rosmasterlocal
rosbag play <bagfile> --clock
```

A.2.5 Autonomous Driving

This section explains the steps necessary to run trained Keras models for autonomous angle prediction on the SPURV.

Installing CUDA Toolkit

If CUDA is not installed on the SPURV, you'll have to do so through Jetpack and the host laptop, as the normal CUDA installation process is not supported for aarch64-architectures. On the SPURV, check what version of L4T (Linux4Tegra) you're running:

```
uname -a
```

If that doesn't give you the release info, try:

```
cat /etc/nv_tegra_release
```

Download and install the matching Jetpack version **on the host laptop** from here: `https://developer.nvidia.com/embedded/jetpack-archive`.

Important! From the installer list, make sure that you set the following options to "no action". Otherwise, they will remove all files from the SPURV and upgrade the operating system:

- Linux for Tegra Host Side Image Setup
- File System and OS
- Drivers
- Flash OS Image to Target

Once the installer completes, CUDA Toolkit should be installed on the SPURV. During the installation, you'll be asked to connect to the SPURV through the installer. Change your network to ResearchSPURV and specify the SPURV's hostname in the installer.

Installing TensorFlow on the SPURV

The SPURV will need to connect to the internet to complete the following steps. Connect a keyboard, mouse and monitor to the SPURV through the ports in the back. Then, follow the steps in Section 3.6.1 in the manual provided by KVS (Section A.3). After changing to `WISP AP` mode and turning off the DHCP server, connect to the internet through Ethernet or WiFi. (You may have to change the network you're connected to in the status bar) Remember to change the network settings back when you're done.

TensorFlow must be installed with `pip` from a `.whl` file. If you're lucky, someone has already built such a file for exactly the same version of Jetpack and Python for an NVIDIA Tegra X2 and uploaded it to the internet. We used the Python 2.7 file from here: `https://github.com/openzeka/Tensorflow-for-Jetson-TX2`.

Install it with

```
pip2 install <wheel file>
```

If you can't find matching files, you'll have to install TensorFlow from source: `https://www.tensorflow.org/install/install_sources`

Installing Keras

This should be fairly problem-free. Simply run:

```
pip2 install keras
```

Using the Autonomous Drive Node

Transfer your trained model to the SPURV through for instance `scp`. Put it in the folder `spurv_research/spurv_examples`. Make sure that the name of the model file matches the one in `autnomous_drive.py` and that the INPUT_SIZE and the other parts of the image preparation match your model.

On the host laptop, connect the Xbox controller and start the `/joy` node:

```
sudo xboxdrv
```

```
rosrun joy joy_node
```

On the SPURV, start the `Autonomous Drive` node. Complete code can be found in Appendix B.3:

```
rosrun spurv_examples autonomous_drive.py
```

The LB and RB buttons on the controller are used to start and stop autonomous mode. The SPURV will not move when the autonomous mode is off.

GPU memory issues

We encountered what turned out to be a memory issue when trying to run larger networks on the SPURV. The error message was "cuDNN launch failure: input shape

[...]". This issue was solved by setting the upper bound on the fraction of GPU memory that is made available to each TensorFlow process to 0.7, as in the code below. Note that this has to be done before importing Keras.

```
import tensorflow as tf
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.7
session = tf.Session(config=config)
```

A.3 User Guide Supplied by KVS Technologies

This section contains the user manual delivered along with the SPURV car by KVS Technologies.

KVS technologies

BRUKERMANUAL

SPURV – Research line

Endringslogg – SPURV Research line – Brukermanual		
Versjon	**Dato**	**Sammendrag av endring**
1.0	18.09.2017	Første utgivelse
1.1	14.12.2017	Oppdatert versjon

1. Faremomenter

1.1. LiPo-batterier

- SPURV inneholder LiPO batteripakker. Dette er batteripakker med høy energitetthet for at enheten skal være lett transportabel, og ha lang batteritid.
- Lad kun enheten under oppsyn. Ta ikke batteriene ut av enheten.
- Det anbefales å benytte medfølgende lader, som balanserer cellespenningen under lading.
- Unngå å tømme batteriet fullstendig ved å skru av enheten med strømbryter etter bruk.
- Skulle det oppstå brann, utløs alarm/varsle brannvesen, og fjern om mulig andre brennbare materialer i nærheten. Fjern deg selv fra brannen dersom du ikke klarer å slukke eller kontrollere den.

2. Oversikt over systemet

2.1. SPURV

SPURV spesifikasjoner	
Operasjonstid	3 timer ved normal bruk
Rekkevidde	Tilkoblet tradisjonell WiFi, omtrent 100m. Opptil 2km ved bruk av ekstern sender
Dimensjoner	500 x 450 x 250 mm uten antenner
Vekt	9 kg
Maksimal hastighet	45 km/t (12 m/s)
Kamera	OnSemi AR0330 3.4MP USB3 1080p60 Full HD lavtlys-kamera – Opsjon: FLiR Machine Vision – Blackfly S ZED Stereo Camera
Varmesøkende-kamera (Standardutgave)	Oppløsning: 80 x 60 Oppdateringsrate: 9 Hz Synsvinkel: 51 ° Termisk sensitivitet: <50 mK Spektrum: 8-14 µm Opsjon: FLiR A35 GigE Compliant FLiR A65 GigE Compliant
Motorkontroller	Feltorientert kontroll (FOC) med Strømtrekksmodus (Torque), Hastighetsmodus(Speed), pådragsmodus(Duty_cycle)
Watchdog	500ms timeout, aktiv brems

2.2. Batterilader

Enheten bruker en lithium-ion lader fra mascot for enklest mulig operasjon og vedlikehold. Laderen er spesielt laget for å kunne lade LiPo-batterier, slik at man forsikrer seg om at lading skjer på en trygg måte. Den kommer ferdig satt opp med standardinstillinger fra fabrikk, og starter lading så snart man kobler den til enheten. Internt i enheten sitter elektronisk batteriovervåkning, som tar seg av cellebalansering og andre sikkerhetskritiske funksjoner.

Ved bruk vil laderen lyse enten Orange, Gult eller Grønt. Orange betyr at bilen lader ved full ladestrøm. Ved gult lys topplades batteriet, og ved grønt lys er batteriet fulladet.

Batterilader	
Spenning inn	100 – 240V AC
Aktiv lading	Orange LED
Topplading	Gul LED
Batteriet er fulladet	Grønn LED

2.3. Grunnleggende vedlikehold

- Sjekk at alle utvendige bolter er på plass og at ingen bolter er løse
- Tørk av fuktighet med en klut for å hindre fare for korrosjon
- Spray metalldeler og skruer i hjuloppheng og nedre ramme med rusthindrende spray for å forhindre korrosjon
- Sjekk at boltene for hjulene sitter fast, og stram til dersom de er løse

Kontakt leverandør dersom andre feil eller mangler oppstår.

3. Bruk av systemet

3.1. Forhåndsinstallert programvare

Spurv kommer forhåndsinstallert med operativsystemet Linux Ubuntu 16.04 (Xenial) og ROS Kinetic Kame, om bord på NVIDIA Jetson TX2 som står i enheten.

ROS Kinetic Kame er en mellomvare, basert på åpen kildekode, som gjør det enklere for utviklere å lage modulær programvare. Det tar hånd om kommunikasjon mellom flere prosesser som kjører på samme datamaskin, og gjør at enkeltprogrammer skrevet i både python og C/C++ kan kommunisere med hverandre, ved at informasjon sendes over ethernet. Dette gjør at systemet ikke bare kan kommunisere på tvers av kodespråk, men også på tvers av flere datamaskiner i et distribuert system.

Vi anbefaler brukere av spurv å sette seg inn i ROS økosystemet, og anbefaler å gjennomgå den offisielle oppstartsguiden her; http://wiki.ros.org/ROS/StartGuide

Sensorverdier og måleenheter brukt i ROS økosystemet følger SI standarden. Det anbefales derfor at programmer man utvikler gjør det samme. For en beskrivelse av typiske måleenheter og koordinatsystemer i bruk, se følgende lenke; http://www.ros.org/reps/rep-0103.html

3.2. Tilkobling til SPURV via SSH

Datamaskinen om bord i enheten er satt opp til å kunne kontrolleres over ssh. For å koble til brukes standardport 22.

Brukernavn	
Passord	

Kommandoeksempel for å logge på via ssh:

ssh brukernavn@192.168.1.5 – Enter - Skriv deretter inn passordet.

Via ssh brukes enheten som om man sitter i kommandolinje på enheten.

3.3. ROS workspace om bord på enheten

Det er allerede lagt inn et workspace for bygging av kode på enheten. Dette ligger under ~/ros/
Det anbefales å ikke opprette et nytt workspace, da enheten er satt til å automatisk starte opp de nodene som trengs for at systemet skal fungere. Dersom man lager et nytt workspace vil oppstarts-script lete etter pakkene på feil område.

For å legge inn ny kode, og lage nye pakker, følg guide på http://wiki.ros.org/catkin/Tutorials/CreatingPackage . Nye pakker legges under ~/ros/src/dittpakkenavn/. For å bygge alle pakker bytt mappe til ~/ros/ og kjør kommando catkin_make i terminal. Dette vil kjøre cmake på C-pakker og lage snarveier til python scripts.

Oppstartsscript

Når enheten skrues på er den instillt på å automatisk starte opp alle nødvendige noder for at systemet skal være klart til bruk.

Oppstartsfilen ligger under /etc/systemd/system/ og kan startes/stoppes vha kommandoene

sudo systemctl stop rosspurv.service
sudo systemctl start rosspurv.service

Alternativt, kan launch-filen til spurv startes fra spurv_launchers pakken, ved hjelp av kommando

roslaunch spurv_launchers spurv.launch

Launch-filen finnes i mappen ~/ros/src/spurv_research/spurv_launchers/launch/ hvor man kan gjøre endringer i instillingene for enheten.

3.4. Tilkobling via HDMI og USB

Enheten er utstyrt med både HDMI og USB på bakplate. Dette gjør at man kan koble til skjerm, og tastatur/mus etter behov. Ved tilkobling via HDMI vil datamaskinen starte opp i grafisk modus, og boote til linux ubuntu's Unity skrivebordsmiljø. Terminalvindu kan åpnes ved hjelp av tastekombinasjonen CTRL+ALT+T, eller via launcher (Øvre venstre hjørne).

Login info

Brukernavn	
Passord	

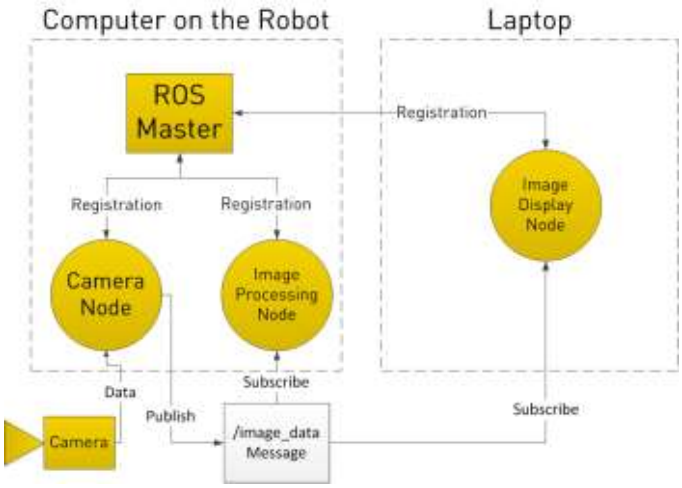
3.5. Installasjon av ROS på egen datamaskin

Det anbefales å bruke Linux Ubuntu 16.04 (Xenial), dersom du ikke har dette, følg guide på https://help.ubuntu.com/16.04/installation-guide/
Deretter følg ROS installasjonsguide for kinetic kame på http://wiki.ros.org/kinetic/Installation .

Legg til følgende variabler i kommandolinje-oppstartfil, ved å åpne filen ~/.bashrc i en tekst-editor,

og legge til følgende tekst;
ROS_MASTER_URI=http://ipaddresse-til-spurv:11311/
ROS_IP=din-ip-addresse

Etter at man har installert ROS på egen datamaskin, og lagt til de nødvendige variablene i shell-environment, kan man hente ut sensorinformasjon i kommandolinje.



Eksempel på å hente ut en liste over tilgjengelig sensordata:

```
rostopic list
```

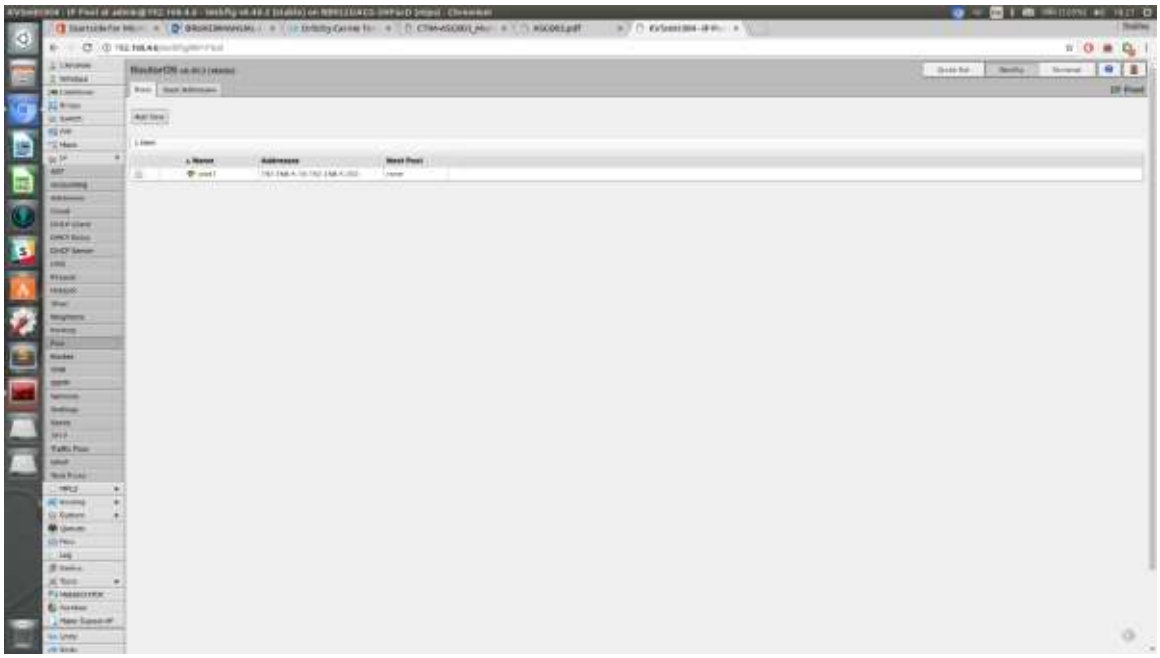
3.6. Nettverksoppsett

3.6.1. Aksesspunkt

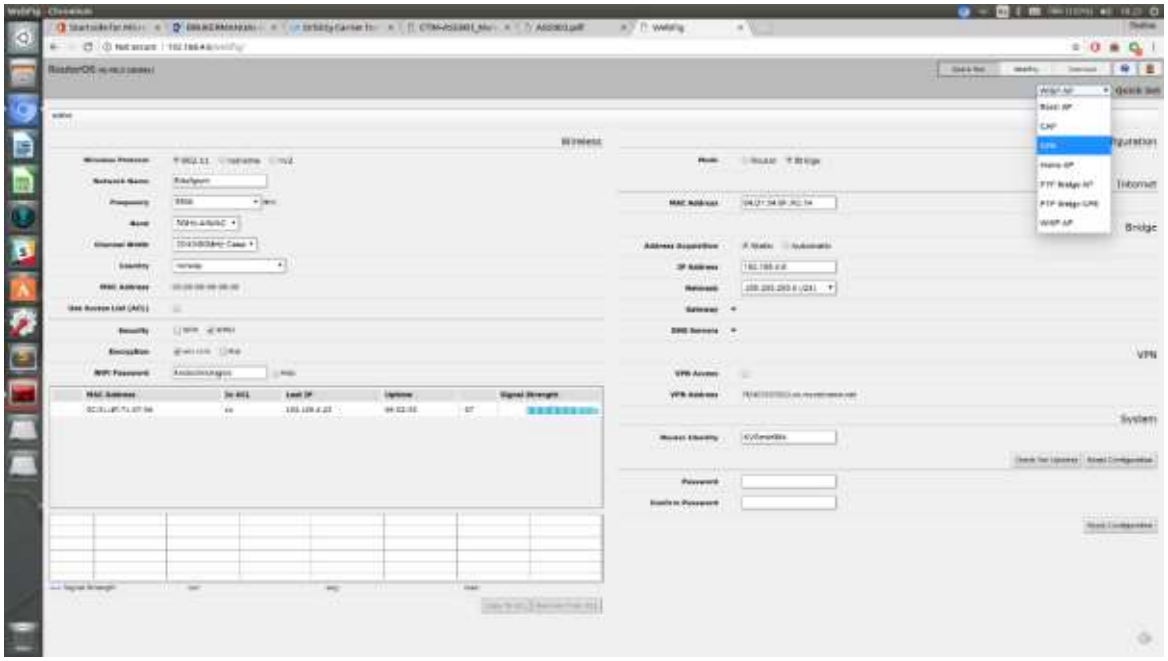
WIFI SSID: "NTNUSpurv"
Passord:
Frekvens: 5ghz (802.11ac)

Enheten kommer satt opp med et wifi aksesspunkt, slik at man raskt kan komme i gang med å arbeide på systemet. Aksesspunktet er satt opp med fast IP-addresse 192.168.1.6 . Ved å skrive inn addressen i nettleser kommer man til konfigurasjonsskjermen. Alternativt kan man også benytte Mikrotik's "winbox" applikasjon for å detektere

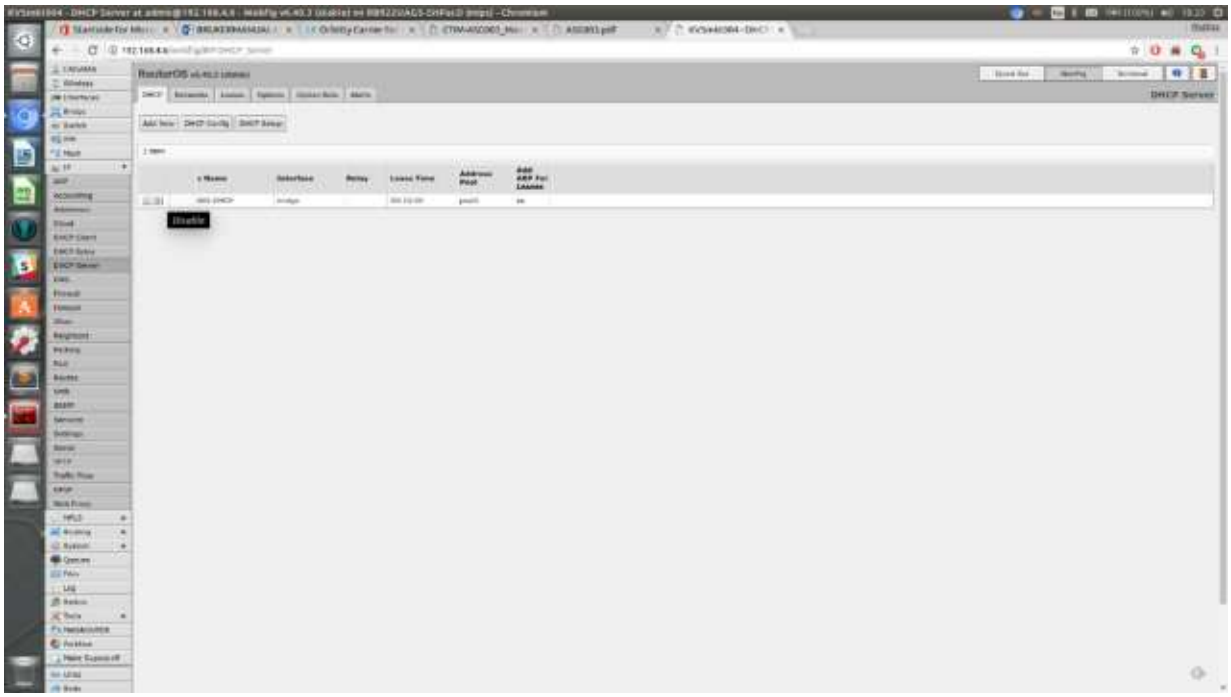
Om bord på aksesspunktet kjører en DHCP-server som tar seg av tildeling av IP-addresser. Tildelingsområde kan endres ved å velge IP, og deretter POOL fra menyen på venstre side. Under pool kan man legge til områder med ip-addresser som blir tildelt.



Det er også mulig å få aksesspunktet til å koble seg på et annet aksesspunkt som klient dersom man ønsker dette. Man må da bytte fra WISP AP (Aksesspunktmodus) mode til CPE mode (Klientmodus).



I CPE mode kan man velge SSID man ønsker at spurv skal koble seg til. Oppsatt DHCP server under punktet IP-> DHCP, må da deaktiveres.



3.6.2. IP-Addresser

Jetson TX2: 192.168.1.5
Mikrotik Aksesspunkt: 192.168.1.6
FLiR Machine vision Front: 192.168.1.3 (Mottar DHCP Addresser)
Eventuelle varmesøkende kamera, samt low-light kamera er koblet til via USB til Jetson TX2, og hentes ut via rqt_image_view/V4L2(Video4Linux2)/gstreamer/opencv, som «USB webkamera».

4. Kjøring av spurv

4.1. Eksempelprogrammer:

Medfølgende enheten kommer eksempelprogrammer som henter styrekommando fra joystick og publiserer dette til de riktige topics og meldingstyper. Eksempelprogrammene kjøres typisk sett på datamaskinen som er koblet til spurv, da de krever joystick-input for å fungere.

Medfølgende eksempler er:

cmd_vel_example.py	Setter kinematiske styrekommandoer XYZ Hastighet Rotasjonshastighet om XYZ
imageconverter.py	Konverterer fra ros image transport til opencv
joystick_example.py	Viser hvordan man setter lav-nivå motorkontroller outputs direkte
opencv_cameras.py	Viser bruken av imageconverter.py

Eksempelprogrammene startes ved å bruke:

```
rosrun spurv_examples filnavn
```

Eventuelt kan man kjøre flere noder samtidig ved hjelp av launch-filer;

```
roslaunch spurv_examples cmd_vel_example.launch
```

4.2. Sending av kommandoer

Styrekommandoer sendes via ROS økosystemet. Ved å publisere informasjon på ROS-Bus vil man kunne sende kommandoer for å skru av og på GPIO, samt utføre styring og motorkontroll. Seksjonen under vil beskrive de forskjellige topics man kan publisere informasjon på for å få spurv til å utføre kommandoer.

4.2.1. /cmd_vel

cmd_vel er et standard topic som brukes av mange roboter som bruker ROS. Dette gjør at mange open source pakker velger å publisere sine kommandoer til dette topicet. Cmd_vel er en høy-nivå metode for å sette hastighet, brukt både for holonomiske, og ikke-holonomiske mobile roboter, og kan derfor inneha mer informasjon enn man trenger for en ackermann-styrt enhet.

Ved å sende en pakke bestående av to 3-dimensjonale vektorer (Geometry_msgs/twist http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html), kan man bestemme ønsket hastighet og retning. Enhetens X er retning fremover, med rotasjonspunkt rundt Z (Positiv rotasjon svinger mot venstre, aksene følger høyrehåndsregelen med Z positiv oppover). Som følge av at enheten ikke beveger seg holonomisk, er translasjonsmeldingene i Y og Z alltid 0. Rotasjon rundt x og y er også 0 til enhver tid. Ved hjelp av noden "cmd_vel_to_ackermann_drive" som ligger i ROS Arbeidsområde ~/ros/src/spurv_ros/cmd_vel_to_ackermann_drive/ konverteres meldingene fra /cmd_vel til /ackermann_cmd.

float64 x	Translasjonshastighet X – Fremover (m/s)
float64 y	Translasjonshastighet Y – Alltid 0
float64 z	Translasjonshastighet Z – Alltid 0
float64 x	Rotasjonshastighet – Alltid 0
float64 y	Rotasjonshastighet – Alltid 0
float64 z	Rotasjonshastighet – Sving (radianer/sec)

4.2.2. /Ackermann_cmd

Dersom en melding publiseres på cmd_vel blir disse automatisk konvertert til Ackermann_cmd (http://docs.ros.org/kinetic/api/ackermann_msgs/html/msg/AckermannDrive.html) meldinger, som nå inneholder informasjon om ønsket setpunkt på hastighet, styrevinkel og tillatt maksimal akselerasjon. Begrensning av maksimal jerk, og steering_angle_velocity er ikke implementert.

Det er også mulig å bruke Ackermann_cmd direkte til å kjøre bilen. Dette gjøres ved å publisere informasjon til topic'et, med meldingens innhold:

float32 steering_angle	# desired virtual angle (radians)
float32 steering_angle_velocity	# Unimplemented, moves as fast as possible
float32 speed	# desired forward speed (m/s)
float32 acceleration	# desired acceleration (m/s^2)
float32 jerk	# Unimplemented, will accelerate at set acceleration value

4.2.3. /Commands/motor/

Driftsmotoren kan styres direkte via tre moduser;
Torque/Duty/Speed basert på hvilket topic man velger å publisere informasjon på. Det må ikke publiseres informasjon på flere av disse samtidig, da dette vil føre til konflikt i motorkontrollerens mykvare. Dersom man velger å kommandere motoren direkte via /commands/motor vil det ikke settes noen form for begrensning av hastighet, akselerasjon eller jerk.

Float64 /Commands/motor/brake	# Brake current setpoint (ampere)
Float64 /commands/motor/current	# Motor current setpoint (ampere)
Float64 /commands/motor/duty_cycle	# Motor duty cycle setpoint (-1 to 1)
Float64 /commands/motor/position	# Not used on SPURV.
Float64 /commands/motor/speed	# Motor speed setpoint (electrical RPM)
Float64 /commands/servo/position	# Steering servo position (Radians)

Dersom man slutter å sende meldinger til motorkontrolleren, vil watchdog timeren slå inn, og motorkontrolleren bremse.

For å publisere ønsket hastighet og styrevinkel kan man teste;

rostopic pub -r 10 /commands/motor/speed -- std_msgs/Float64 2000
rostopic pub -r 10 /commands/motor/speed -- std_msgs/Float64 -2000 (bakover)
rostopic pub -r 10 /commands/servo/position -- std_msgs/Float64 0.65 (NB! Ved setting av direkte styrevinkel må ikke verdier under 0.35 og over 0.65 brukes da dette kjører servoene inn i mekanisk endestopp – dette taes automatisk hensyn til ved bruk av ackermann_cmd eller cmd_vel)

-r 10 brukes for at det skal sendes meldinger med 10hz oppdateringsfrekvens.

Se også eksempelprogram: joystick_example.py for hvordan dette gjøres ved hjelp av Python.

4.2.4. ROS servicecall for setting av GPIO

De fire GPIO'ene kan settes individuelt ved å sende et servicecall.
For å liste servicecalls som er tilgjengelig bruk kommando:

Rosservice list

Følgende service call er tilgjengelige for styring av gpio-en.

/spurv_gpio_node/get_gpio_state	Henter nåværende logisk nivå for en gitt utgang eller inngang fra 0-4, gir svar 0 eller 1.
/spurv_gpio_node/set_gpio_state	Setter logisk nivå på en gitt utgang. Verdi mellom 0 og 1.
/spurv_gpio_node/set_gpio_direction	Setter retning på en gitt gpio, om den skal være input eller output. Verdi "out" eller "in".
/spurv_gpio_node/toggle_gpio	Veksler mellom høy og lav verdi på en gitt utgang.

Merk: Skal en lese korrekt logisk verdi på en gpio, hvor er det viktig at direction/retning er satt som "in" hvis en ønsker å lese en verdi som kommer inn. Skal en styre en perifærmodul med pinout konfigurert som utgang, må en sette retning til "out".

Eksempel service call for setting av utgang 0 til høy:

```
Rosservice call /spurv_gpio_node/set_gpio_state 0 1
```

4.3. Tilgjengelige sensordata på ROS-Bus

Alle tilgjengelige sensordata blir publisert til ROS økosystemet.

For å liste tilgjengelig informasjon bruk kommando:

```
rostopic list
```

For å se informasjon som ligger på BUS, bruk kommando;

```
rostopic echo «/valgt/topic»
```

4.3.1. /Sensors/core

Sensors/Core inneholder informasjon fra motorkontrolleren.

float64 voltage_input	input voltage (volt)
float64 temperature_pcb	temperature of printed circuit board (degrees Celsius)
float64 current_motor	motor current (ampere)
float64 current_input	input current (ampere)
float64 speed	motor electrical speed (revolutions per minute)
float64 duty_cycle	duty cycle (0 to 1)
float64 charge_drawn	electric charge drawn from input (ampere-hour)
float64 charge_regen	electric charge regenerated to input (ampere-hour)
float64 energy_drawn	energy drawn from input (watt-hour)
float64 energy_regen	energy regenerated to input (watt-hour)
float64 displacement	net tachometer (counts)
float64 distance_traveled	total tachnometer (counts)
int32 fault_code	0-None, 1- Over-Voltage, 2-Under Voltage, 3-DRV Fault, 4-Over-current, 5Over-Temp-FET. 6 Over-Temp Motor

4.3.2. /Odom

Publiserer informasjon om odometri fra bilen. Verdiene er estimert ved å beregne hastigheten på motor om til SI-enheter. Posisjon integreres med hensyn til tid fra hastighet, basert på sin/cos av styrevinkel og hastighet. Parameterene som styrer omregning fra rpm til m/s ligger i config-mappe, under **~/ros/src/vesc/vesc_ackermann/cfg.** Disse parameterene kan tunes etter ønske.
Informasjon om pakkeinnhold:
http://docs.ros.org/api/nav_msgs/html/msg/Odometry.html

4.3.3. Innhenting av bildedata/ videostrøm

Videostrøm kan innhentes via ROS image transport (http://wiki.ros.org/image_transport). Det er også lagt ved eksempelkode for innhenting av video til opencv i eksempelprogrammene. Se ~/ros/src/spurv_research/spurv_examples/

Videostrøm ligger på topics:
fwd_camera/image_raw/
bwd_camera/image_raw/
thermal_camera/image_raw #Hvis varmesøkende kamera

4.4. Logging og avspilling av logget data

Ved å bruke "rosbags" kan man logge data fra kjøringen. Dataene kan spilles tilbake på et senere tidspunkt, hvor data som var tilgjengelig under loggingen vil bli publisert som om det kommer fra bilen.

Eksempel på lagring av all sensordata mens man kjører, samt avspilling:

rosbag record -a
rosbag play recorded1.bag

Dokumentasjon av bruken av rosbags kan finnes her: http://wiki.ros.org/rosbag og http://wiki.ros.org/rosbag/Commandline

5. Konfigurering av noder

Systemet er bygget opp av flere noder, som er enkle programmer som kjører selvstendig og publiserer data ut i systemet. Hver node har ofte parametre en kan sette som bestemmer hvordan nodene oppfører seg. Eksempelvis kan en konfigurere kamera til å kjøre forskjellige oppløsninger eller bildeformat.

5.1. Konfigurering av kamera

Det sitter 2 kamera på spurven, ett i front og ett bak-kamera. Disse kameraene kan konfigureres for å kjøre ulike typer bildeformat, hvor mange bilder per sekund (FPS) overføringen skal ha eller hvilken oppløsning bilde skal ha. Alle kameraene på spurv-en konfigureres på samme måte ved å endre parameterene i launch-filen til kameranodene.

De aktuelle konfigurasjonsfilene til kameraene som følger med installasjonen på spurv-en finnes i ros pakken ***spurv_camera*** og ligger *i* ***launch*** mappen.

5.1.1. Tilgjengelige paramtere

Kameraene har tilnærmet samme parametere, men det er noen kamera som ikke støtter samme oppløsning eller som har andre ulikheter. Under ligger en liste med parametere som er tilgjengelige for den respektive kameramodellen.

FLIR Blackfly S

Parameter	***Eksempelverdi***	***Verdiområde***	***Beskrivelse***
serial	"FLIR-17000486"	Valgfri string	Serienummeret til kameraet en ønsker å koble seg til
framerate	30	1 – 78.	Antall bilder per sekund.
resolution	*"1440x1080"*	*Valgfri, bilde blir kuttet hvis ikke det er maks.*	Oppløsning på kameraet.
videoformat	*"bayer"*	*"gray8", "rgb", "bayer".*	Hvilket bildeformat en ønsker fra kamera.

Merk: Videoformat bestemmer hvilket format en skal hente inn fra kamera. Bayer er mest effektivt hvis en ønsker farge, men en del prosessering må skje på tx2 modulen for å konvertere bayerformatet til rgb. Derfor er "rgb" også mulig å hente direkte fra kamera, men dette krever veldig høy båndbredde og det er ikke mulig å oppnå mer enn 24 fps. Svarthvitt er mest effektivt, men er selvfølgelig ikke ett alternativ dersom brukeren ønsker farge.

Econ Systems

Parameter	***Eksempelverdi***	***Beskrivelse***
device	/dev/video0	Fysisk filnavn til kamera i linux.
framerate	30	Antall bilder per sekund
resolution	*"1280x720"*	Oppløsning på kameraet.

Merk: her er ikke videoformat tilgjengelig. Econ kamera støtter kun UYVY format eller YUV 422 som det også heter.

Merk: Med dette kamera, støttes kun ett gitt antall kombinasjoner på oppløsning og framerate. Sjekk tabellen.

Oppløsning	**Framerate**
640 x 480	60 & 45
960 x 540	58 & 30
1280 x 720	45 & 30
1280 x 960	34
1920 x 1080	30 & 15

Appendix B

Code and Scripts

This appendix contains various scripts related to this thesis, with emphasis on data processing and training. The full code can be found in the attached .zip-file as well as on `www.github.com/ragnhildneset/MasterThesis`.

B.1 Docker Setup

Docker must be used on the machines in the Visual Computing Lab at NTNU. Our Docker image references the official TensorFlow image with GPU support. It is defined through the following Dockerfile:

Dockerfile

```
FROM gcr.io/tensorflow/tensorflow:latest-gpu

# install dependencies from debian packages
RUN apt-get update -qq \
 && apt-get install --no-install-recommends -y \
    libsm6 \
    libxext6 \
    libfontconfig1 \
    libxrender1 \
    python-tk \
    python-pillow \
    vim

# install dependencies from python packages
RUN pip --no-cache-dir install \
    opencv-python \
    seaborn \
    keras \
    keras-vis
```

```
# install your app
RUN mkdir -p /ai

WORKDIR /ai

CMD ["/bin/bash"]
```

To build the image:

```
nvidia-docker build -t annaragnhild-tensorflow .
```

To start the container, run the command:

```
nvidia-docker run -it -v "\$(pwd)":/ai -p 8888:8888
   annaragnhild-tensorflow
```

B.2 VisualBackProp

This sections includes the complete code for the VisualBackProp implementation. Example usage:

```
from matplotlib import pyplot as plt
from visual_backprop import VisualBackprop

visual_backprop = VisualBackprop(model) # Keras model

# image is a single preprocessed image
mask = visual_backprop.get_mask(image)[:, :, 0]

# values for colormap normalization
vmin = np.min(mask)
vmax = np.percentile(mask, 99)

plt.figure()

# show as overlay on image
plt.imshow(image)
plt.imshow(mask, alpha=.6, cmap='jet', vmin=vmin, vmax=vmax)

# can also be shown as a black and white mask:
plt.imshow(mask, cmap='gray', vmin=vmin, vmax=vmax)
```

visual_backprop.py

```
'''
    Generalized version of VisualBackprop implementation by:
    https://github.com/experiencor/deep-viz-keras
    Which only supported vgg16
'''

import numpy as np
import keras.backend as K
from keras.layers import Input, Conv2DTranspose
from keras.models import Model
from keras.initializers import Ones, Zeros


class VisualBackprop():
    """
        Computes a saliency mask with VisualBackprop
        https://arxiv.org/abs/1611.05418
    """

    def __init__(self, model, output_index=0):
        inps = [model.input, K.learning_phase()]          # input
           placeholder
        outs = [layer.output for layer in model.layers]   # all
           layer outputs
        self.forward_pass = K.function(inps, outs)        #
           evaluation function

        self.model = model

    def get_mask(self, input_image):
        """
            Returns a VisualBackprop mask for an image.
        """
        x_value = np.expand_dims(input_image, axis=0)
        visual_bpr = None
        layer_outs = self.forward_pass([x_value, 0])

        conv_layers = []
        for layer, layer_out in zip(self.model.layers, layer_outs):
            if 'Conv2D' in str(type(layer)):
                conv_layers.append((layer, layer_out))

        # Iterate over feature maps upstream
        for i in range(len(conv_layers)-1, -1, -1):
            # Average the feature map
            layer = np.mean(conv_layers[i][1], axis=3,
               keepdims=True)

            if visual_bpr is not None:
                if visual_bpr.shape != layer.shape:
                    visual_bpr = self._deconv(visual_bpr,
                                              conv_layers[i+1][0])
                    # If upsampling fails
                    if visual_bpr.shape != layer.shape:
                        visual_bpr =
```

```
                        self._add_padding_to_match_shape(
                                                visual_bpr,
                                                layer.shape)
                visual_bpr = visual_bpr * layer  # Pointwise
                    product
            else:
                visual_bpr = layer

        # Last upsampling to input image size
        visual_bpr = self._deconv(visual_bpr, self.model.layers[0])
        visual_bpr = self._add_padding_to_match_shape(
                            visual_bpr, x_value[:, :, :, :1].shape)
        return visual_bpr[0]

    def _add_padding_to_match_shape(self, visual_bpr,
        target_shape):
        """
            Add one pixel of padding in top and left side of mask
                if the deconv
            operation doesn't upsample to the exact size of
                upstream layer.
            (If it fails, it is usually off by one pixel less than
                wanted)
        """
        # Check difference between upsampled feature map shape and
            wanted
        # shape in every dimension
        diff = [target_shape[i] - visual_bpr.shape[i]
                for i in range(len(visual_bpr.shape))]
        for axis in range(len(diff)):
            if diff[axis] > 0:
                visual_bpr = np.insert(visual_bpr, 0, 0, axis=axis)
        return visual_bpr

    def _deconv(self, feature_map, upstream_feature_map):
        """
            The deconvolution operation to upsample the average
            feature map upstream.
        """
        x = Input(shape=(None, None, 1))
        y = Conv2DTranspose(filters=1,
                            kernel_size=upstream_feature_map.kernel_size,
                            strides=upstream_feature_map.strides,
                            kernel_initializer=Ones(),
                            activation=upstream_feature_map.activation,
                            use_bias=upstream_feature_map.use_bias,
                            bias_initializer=Zeros())(x)

        deconv_model = Model(inputs=[x], outputs=[y])

        inps = [deconv_model.input, K.learning_phase()]   # input
            placeholder
        outs = [deconv_model.layers[-1].output]           # output
            placeholder
```

```
        deconv_func = K.function(inps, outs)                #
            evaluation function

        return deconv_func([feature_map, 0])[0]
```

B.3 Autonomous Driving Node

This sections includes the complete code for the Autonomous Driving Node implementation. Usage is covered in Appendix A.2.5

visual_backprop.py

```
#!/usr/bin/env python

import rospy
import cv2
import numpy as np
from keras.models import load_model
from sensor_msgs.msg import Joy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
from ackermann_msgs.msg import AckermannDriveStamped

DISABLE_AUTONOMOUS=4 #=LB-button on xbox-controller
ENABLE_AUTONOMOUS=5 #=RB-button on xbox-controller
SPEED = 0.4 # m/s

MODEL = "model.h5"
INPUT_SIZE = (66,200,3) # Input size of the model
CUT_RATIO = 0.65 # For cutting off the top of the image

frame_id = rospy.get_param('~frame_id', 'odom')
max_accel_x = rospy.get_param('~acc_lim_x', 1.0)
max_jerk_x = rospy.get_param('~jerk_lim_x', 0.0)

class RunModel(object):

    def __init__(self):
        self.autonomous_mode = False
        self.cv_bridge = CvBridge()
        self.init_model()
        self.init_pub_sub()
        rospy.loginfo("Spurv␣autonomous␣driver␣initialized")

    def init_model(self):
        self.model = load_model(MODEL)
        # A dummy-prediction is needed initialize the model
            properly:
        self.model.predict(np.zeros((1, INPUT_SIZE[0],
            INPUT_SIZE[1], INPUT_SIZE[2])))
```

```
    def init_pub_sub(self):
        self.image_subscriber =
           rospy.Subscriber("/fwd_camera/image_raw", Image,
           self.on_image_callback)
        self.joystick_subscriber = rospy.Subscriber("/joy", Joy,
           self.on_joy_callback)
        self.steering_publisher =
           rospy.Publisher("/ackermann_cmd",
           AckermannDriveStamped, queue_size=1)

    def on_image_callback(self, data):
        if self.autonomous_mode:
            image = self.cv_bridge.imgmsg_to_cv2(data, "bgr8")

            # Prepare image
            offset = int(image.shape[0] - (image.shape[0] *
               CUT_RATIO))
            image = image[offset:offset + image.shape[0],
               0:image.shape[1]] # Cut off the top (1-CUT_RATIO)
               part of image
            image = cv2.resize(image, (INPUT_SIZE[1],
               INPUT_SIZE[0])) # Reduce to input size
            image = image / float(255) - 0.5 # Normalize

            # Predict steering and publish
            angle = self.model.predict(np.array([image]))[0][0]
            self.publish_steering(angle)

    def on_joy_callback(self, joyMessage):
        enable = joyMessage.buttons[ENABLE_AUTONOMOUS]
        disable = joyMessage.buttons[DISABLE_AUTONOMOUS]
        if bool(enable):
            self.autonomous_mode = True
            print "Autonomous mode enabled"
        if bool(disable):
            self.autonomous_mode = False
            print "Autonomous mode disabled"

    def publish_steering(self, angle):
        if self.autonomous_mode:
            msg = AckermannDriveStamped()
            msg.header.stamp = rospy.Time.now()
            msg.header.frame_id = frame_id
            msg.drive.steering_angle = angle
            msg.drive.speed = SPEED
            msg.drive.acceleration = max_accel_x
            msg.drive.jerk = max_jerk_x
            self.steering_publisher.publish(msg)


if __name__ == '__main__':
    rospy.init_node("autnomous_driver")
    driver = RunModel()
    try:
        rospy.spin()
```

```
    except KeyboardInterrupt as interrupt:
        pass
```

B.4 Point Angle Path Visualisation

This sections explains the implementation of the car path visualisation described in Section 3.4.1.

Defining the rotation matrix $R(\theta)$ and the angle scaling factor f_θ:

```
import math

# Rotation matrix
def R(theta):
    cos_theta = math.cos(theta)
    sin_theta = math.sin(theta)

    return np.ndarray(buffer=np.array([cos_theta, -sin_theta,
                                       sin_theta, cos_theta]),
                                       shape=(2,2))


def angle_scaling(theta):
    return theta * 0.093
```

Various constants:

```
H = 0.23 # m - distance from camera to ground
D_0 = 0.44 # m - distance from bottom of car to first point on
   ground in camera view
Y_IRL = 0.57 # m - height of image frame at D_0 in real life
X_IRL = 0.4 #1.01 m - width of image frame at D_0 in real life -
   decreased due to wrong scaling

IMAGE_HEIGHT = 200 # pixels - height of displayed image
IMAGE_WIDTH = 300 # pixels - width of displayed image

NOF_DOTS = 30 # the number of dots to display
DRAW_EVERY_NTH_DOT = 1 # draw every n'th dot
FRAME_RATE = 15 # the frame rate, should be the same as capture
   rate
CROPPING_PERCENTAGE = 0.65 # the percentage of the image that
   remains after cropping the top
```

Calculating the $\mathbf{p_i}$ points:

```
import numpy as np

def calculate_axes(predictions, steering):
    h = []
    sum_d = D_0

    for prediction, steering in zip(predictions, steering):
        d_i = steering[1] / float(FRAME_RATE/DRAW_EVERY_NTH_DOT) #
            distance travelled in 1/15th second
        sum_d += d_i
        h_i = H*d_i / float(sum_d)
        h.append(h_i)

    r = []
    r_star = [np.array([0, h[0]])]
    r_1 = np.matmul(R(angle_scaling(predictions[0])), r_star[0])
    r.append(np.array(r_1))
    p = np.zeros(shape= (len(h), 2))
    p[0] = np.array(r_1)

    for i in range(1, len(h)):
        r_i_star = r[i-1] * (h[i - 1] + h[i]) / float(h[i-1]) -
            r[i-1]
        r_star.append(r_i_star)

        r_i = np.matmul(R(angle_scaling(predictions[i])),
            r_star[i])

        r.append(r_i)
        p_i = p[i - 1] + r[i]
        p[i] = np.array(p_i)

    return p[:,0], p[:,1]
```

Scaling the $\mathbf{p}_i$ vectors to $\mathbf{p_i}^{IRL}$:

```
def scale_vector(x_axis, y_axis, to_width, to_height):
    return x_axis * to_width / X_IRL, y_axis * to_height *
        CROPPING_PERCENTAGE / (Y_IRL)
```

Putting it all together to calculate the path coordinates for the ground truth and predictions:

```
def get_point_angle_data(validation_data, predictions,
    starting_point):
    X = starting_point
    indices = np.arange(X, X + NOF_DOTS*DRAW_EVERY_NTH_DOT,
        DRAW_EVERY_NTH_DOT)
```

```
    preview_image = cv2.imread(os.path.join(dataset_path,
       valid_data['image_names'][indices[0]]))
    preview_image = cv2.resize(preview_image, (IMAGE_WIDTH,
       IMAGE_HEIGHT))

    wanted_predictions = pred[indices]

    x_axis, y_axis = calculate_axes(wanted_predictions,
       validation_data['steers'][indices])
    true_x_axis, true_y_axis =
       calculate_axes(validation_data['steers'][:,0][indices],
       validation_data['steers'][indices])

    x_axis, y_axis = scale_vector(x_axis, y_axis, IMAGE_WIDTH,
       IMAGE_HEIGHT)

    x_axis = x_axis + IMAGE_WIDTH / 2
    y_axis = np.negative(y_axis) + IMAGE_HEIGHT

    true_x_axis, true_y_axis = scale_vector(true_x_axis,
       true_y_axis, IMAGE_WIDTH, IMAGE_HEIGHT)
    true_x_axis = true_x_axis + IMAGE_WIDTH / 2
    true_y_axis = np.negative(true_y_axis) + IMAGE_HEIGHT

    return x_axis, y_axis, true_x_axis, true_y_axis, preview_image
```

Displaying the calculations as paths using `matplotlib`:

```
import matplotlib.pyplot as plt

def show_paths(x_axis, y_axis, true_x_axis, true_y_axis,
   preview_image):
    plt.figure(1)
    f, axarr = plt.subplots(1,2, figsize=(15,15))
    preview_image = cv2.cvtColor(preview_image, cv2.COLOR_BGR2RGB)
    axarr[0].imshow(preview_image)
    axarr[1].imshow(preview_image)

    axarr[0].set_title('Predicted␣angles')
    axarr[1].set_title('True␣angles')

    axarr[0].scatter(x=x_axis, y=y_axis, label='Predicted')
    axarr[1].scatter(x=x_axis, y=y_axis, label='Predicted')
    axarr[1].scatter(x=true_x_axis, y=true_y_axis, label="Ground␣
       Truth")
    axarr[1].legend(loc='upper␣right')
```

Usage example:

```
starting_points = range(0, 700, 5)
```

```
for starting_point in starting_points:
    x_axis, y_axis, true_x_axis, true_y_axis, preview_image =
       get_point_angle_data(valid_data, pred, starting_point)
    show_paths(x_axis, y_axis, true_x_axis, true_y_axis,
       preview_image)
```