EXAMENSARBETE INOM MASKINTEKNIK,
AVANCERAD NIVÅ, 30 HP
STOCKHOLM, SVERIGE *2017*

Analysis and Evaluation of Recurrent Neural Networks in Autonomous Vehicles

GABRIEL ANDERSSON SANTIAGO

MARTIN FAVRE

KTH
SKOLAN FÖR INDUSTRIELL TEKNIK OCH MANAGEMENT

KTH Industriell teknik och management

Master of Science Thesis MMK2017:152 MDA604
Analysis and Evaluation of Recurrent Neural Networks in Autonomous Vehicles
Gabriel Andersson Santiago
Martin Favre

Approved :	Examiner: De-Jiu Chen	Supervisor: Mahmood Reza Khabbazi
	Commissioner: Cybercom	Contact Person: Bartoz Libner

Abstract

Once upon a time cars were driven by the pure will and sweat of decent humans. Today technology has reached the point in which complex systems can drive the car with little or no human interaction at all. Whilst it does take away the sweet Sunday drive, one has to consider the positives. Over 90% of all vehicle accidents can be credited to the driver. City traffic can be optimised to avoid congestion. Additionally extending the morning nap to the car ride to work is truly something to strive for. One of the way autonomous driving can be achieved is through Artificial Neural Networks. These systems teaches a model how do drive a car through vast and vast amounts of data consisting of the state and the correct action. No manual logic required! One of the many issues these systems face is that the model only analyses the current state and has no inherent memory, just a million small independent decisions. This creates issues in situations like overtaking as it requires a longer plan to safely pass the other vehicle.

This thesis investigates utilising the Recurrent Neural Networks which are designed to analyse sequences of states instead of a single one with hopes that this may alleviate the sequential hassles. This is done by modifying an 1/12 scale RC-car by mounting a camera in the front. The images were used to control both steering or velocity in three separate tests which simulates normal driving situations in which the sequence of events contain information.

In all three scenarios three different networks were tested. One ordinary single-state model, a model evaluating 5 states and model evaluating 25. Additionally as a ground truth a human drove the same tests. These were qualitatively compared and evaluated.

The test results showed that there indeed sometimes were an improvement in utilising recurrent neural networks but additional and more repeatable tests are required to define when and why.

KTH Industriell teknik
och management

Examensarbete MMK2017:152 MDA604

Analys och Evaluering av ’Recurrent’ Neurala Nätverk i Autonoma Fordon

Gabriel Andersson Santiago
Martin Favre

Godkänt:	Examinator: De-Jiu Chen	Handledare: Mahmood Reza Khabbazi
	Uppdragsgivare: Cybercom	Kontaktperson: Bartoz Libner

Sammanfattning

Traditionellt har bilar körts av anständiga människor. Teknologin har idag dock kommit till den punkten då komplexa system kan köra med minimal eller full avsaknad av mänsklig interaktion. Medans det visserligen tar bort then trevliga söndagsturen så måste man tänka på fördelarna. I över 90% av alla fordonsolyckor är orsaken grundad i föraren. Stadstrafik kan bli optimierad för att undvika trafikstockningar. Dessutom att förlänga ens morgonlur med hela bilresan till jobbet är verkligen något att sträva efter. Ett av sätten man kan uppnå autonom körning är genom artificiella neurala nätverk. Dessa system lär en modell hur man kör med hjälp av stora mängder data som består av ett tillstånd och dess korrekta handling. Minimal mängd manuell design krävd. En av de flera problem som Artificiella Neurala Nätverk har är att de inte har något minne, utan tar bara en stor mängd individuella beslut. Detta kan skapa problem i situationer som omkörning då det kräver en längre plan för att säkert ta sig runt den andra bilen.

Den här uppsatsen undersöker ’Recurrent’ Neurala Nätverk som är designade för att analysera sekvenser av tillstånd istället för en enkelt tillstånd med hopp om att det kommer lindra de sekventiella problemen. Detta är gjort genom att modifiera en 1/12 i skala radiostyrd bil med en kamera på framsidan. Dessa bilder används för att kontrollera både styrning eller hastighet i tre separata experiment som simulerar vanliga körningsscenarion i vilka sekvensen av tillstånd innehåller information.

I alla tre experiment testades tre olika nätverk. Dessa analyserar respektive 1, 5 och 25 tillstånd. Utöver dessa gjordes även experiment med en mänsklig förare som grundreferens. Resultaten jämfördes och evaluerades kvalitativt.

Slutresultaten visade att det fanns tillfällen då det var bättre att analysera flera tillstånd, men att fler och mer repeterbara tester behövs för att kunna slå fast när och varför.

Acknowledgement

Firstly we would like to thank our industrial supervisors at Cybercom, Bartosz Libner and Karl Lunden for keeping us on the right path and adding brilliant insight in this endeavour.

Additionally great thanks to our academical supervisor Mahmood Khabazzi for his expertise in the field as a researcher giving us many a fruitful discussions.

Finally a thanks for our colleagues for keeping us sane.

- Gabriel Andersson Santiago and Martin Favre

Nomenclature

Abbreviations

ANN	Artificial Neural Network
BLSTM	Bidirectional Long Short Term Memory
CAD	Computer-aided Design
CSI	Camera Serial Interface
CNN	Convolutional Neural Network
DARPA	Defense Advanced Research Projects Agency
GPS	Global Positioning System
GPU	Graphics Processing Unit
IMU	Inertial Measurement Unit
LSTM	Long Short Term Memory
NN	Neural Network
PWM	Pulse-width Modulation
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
ROS	Robotic Operating System

Notations

Symbol	**Description**
h_t	Hidden state
W_y	Weight matrix
b_y	Bias
v	Velocity of vehicle
θ	Angle of vehicle
v_θ	Angular velocity of vehicle

Contents

Chapter 1

Introduction

1.1 Background

Autonomous driving is a field of research directed towards creating vehicles that will manoeuvre along roads towards a set destination while obeying traffic rules and avoiding unwanted collisions. The idea of removing the human from control has a multitude of reasons often around comfort, however a more hands-on reason is driving safety. Removing the human factor would be a strong step towards improved safety for driving in motor vehicles. Another opportunity would be the ability to optimise traffic flow by allowing the vehicles to communicating their speed and trajectories to each other and calculate an optimal plan of action.

The idea of letting the car drive itself has existed for a long time and research has given a multitude of tools to assist in driving. Automatic transmission and cruise control are examples driving assistance where mundane tasks are delegated to the vehicle. These are common and can be widely found on the consumer market today. The goal of a fully autonomous car which required no human input to function has not yet been fulfilled as of today. However multiple methods has been researched and tried.

The state of the art approach is to use a combination of sensors such as lidar, radar, GPS and camera to navigate through thoroughly hand-engineered methods. This meditated approach was popular in the 2007 DARPA urban challenge.

Another method is to use machine learning - specifically neural networks - to map an image from a mounted camera directly to a vehicle control action. This end-to-end method tries to emulate the natural way a human drives. These systems generally holds a simpler architecture and there is no need to manually define things like road models or driving situations. The method was presented as early as 1989 by DA. Pomerleau and whilst it showed promising result with the limited hardware the method did not become the most common usage.

In recent years neural networks has resurfaced in form of convolutional neural networks (CNN) as a very effective method to classify images, outperforming most previous methods in success rate. This method extracts features from an image and pools new, smaller images which intensifies these features and makes their position in the image less important. This makes it less sensitive to noise such as translations, rotations and distortions. In relation the interest of using a neural network for end-to-end systems has resurfaced as well.

One issue with the end-to-end method is that it only acts upon the current information and hold no short-term memory for what it just did. This will create issues in situations where a longer term plan is required as well in situations which may look equal in a still image but differs majorly if put into a sequence. A situation is when a driver performs a lane change. The driver may choose to be satisfied with the situation and stay behind the leading vehicle. The driver may also choose to overtake. This both creates conflicting training data and if the car wants to overtake it will have to keep that decision for a longer period of time to give the car time to shift lanes.

Recurrent neural network(RNN) is an architecture that takes its previous decisions back in again and looks at sequences of events. This method has been successful in tasks such as classifying videos. It is possible that RNN would be possible to use in end-to-end systems in autonomous driving to good effect.

1.2 Purpose

This thesis examines many-to-one recurrent neural networks when implemented in an end-to-end architecture on a 1/12 scale remote controlled (RC) car in three different scenarios. Its performance will be compared to a traditional single-state network using a human driver as reference.

Research Questions

The thesis will attempt to answer the following questions.

- Is the recurrent neural network a viable alternative to the traditional single-state network?
- Is there a behavioural difference between the recurrent neural network and the single-state network?
- Is the sequential model more successful in certain scenarios?
- What is the difference in hardware requirements for a recurrent neural network and a single-state network?
- Is the recurrent neural network capable of executing a longer 'plan' or trajectory?

1.3 Scope

A few points points have to be brought up on the limitations this thesis has had in consideration.

- The aim is not to create a fully functional autonomous car but to test specific scenarios.
- The environment is not a perfectly controlled one
- An RC-car does not perfectly model a consumer-grade personal car, however the pros of being a more friendly working environment as well as cheap outweighs the cons.
- The velocity of the RC-car will be kept as such a low rate that slippage and other related physical factors may be ignored.
- Software will be limited to what is available, this thesis uses Keras, a high-level neural network framework, with Python as a result. The algorithms used are limited to what is available in the framework and the performance is limited to the framework, the written code and the hardware available.

1.4 Method

The method used is a qualitative investigation of the two models through experimental design and experimental evaluation.

The first part of the thesis is to do a literature review to learn what has been done in the field of autonomous vehicles and neural networks. It is also done to get a deeper knowledge on how convolutional and recurrent neural networks work, their advantages and limitations. After the literature review development of the neural networks and the vehicle will be done parallel. The work process for the vehicle will be an agile development method with first developing the bare minimum functionality of the vehicle and by iteration adding more functionality. The work process for the neural networks will be similar; first understand how the chosen framework works followed by deciding how the training data should be processed to able to train the networks. The trained networks will then be tested on different scenarios and data will be gathered for evaluation. There is no clear definition from every situation on what is a good drive. This needs to be evaluated based on the ground truth which is the measured driving of a human.

1.5 Ethical considerations

The planned experiments will not cause any personal, social or environmental harm. The hardware will be reused from previous projects or if bought in it will be used in future projects. The RC-car has been geared down and will not achieve the speed where it may be seriously harmful. Voltages will not exceed 9V and are therefore deemed to not be harmful.

The results of this paper does not lead to ethically challenging changes.

Chapter 2

Literature Review

2.1 Autonomous driving

Intro

Self-driving cars holds many potential advantages if successfully implemented. In 2008 approximately *93%* of motor vehicle crashes in USA were deemed attributable to the driver [1]. This number could be greatly reduced to nearly eliminated if the human factor was replaced with an artificial system. However the number of crashes attributable to software errors has potential to rise in the same shift.

The topic of autonomous vehicles has been discussed in different points in the the 20th century, example is Bel Gedde's Magic Motorways in which he foresees that the cars of 1960 will stop the driver from committing dangerous errors and assisting the driver through difficult conditions [2].

Known Methods

Autonomous driving can be implemented in many different ways but they may be categorised in two options depending on what determines the vehicles output. One of these methods is using machine learning to directly map sensor input to vehicle control.

DA. Pomerleau worked on an end-to-end approach on ALVINN which took an input image and a laser range finder as input to the three to four hidden layer neural networks and had steering as output. The picture was 30x32 pixels and the car's top speed was 90 km/h. The training data came from recording a human driver and pairing an image of the road ahead with the drivers steering at the time. This system seemed to function well under the specific conditions it was trained for. It faced multiple issues. If the training data was too homogeneous which made the system only "remember" the most common parts of the training set. There did not exist training data of the vehicle correcting its own faults such as drifting of the road as this would also collect data of the driver doing these mistakes. He also notes the lack of temporal correlation in the sensor data [3].

Over two decades later NVIDIA corporation published a paper using similar end-to-end methods now using a much larger CNN on specialised hardware. They successfully manage to navigate along roads at a claimed 98% of the time. However they do not note any of the issues DA. Pomerlau had. Additionally they don't specifically test their vehicle in scenarios where this would come into effect. It is unclear whether these issues are solved or ignored. It has to be noted that the hardware used in this paper is for sale by the same corporation [4].

Another propose to an end-to-end system is done by Chen *et al* where they attempt to overcome the issue that end-to-end systems only take many low-level decisions without any larger plan. They propose that the system decides on different states such as changing a lane which will inform future decisions on what it's currently doing. Here they also note the issue with lack of a temporal aspect in the neural network [5].

A different approach than the end-to-end is a more meditated approach in which the sensing, interpretation and decision-making are separated.

In the 2007 DARPA challenge this was the predominant method among the finishing vehicles. For sensors they mainly used radar, GPS, camera and multiple lidars. The lidars scanned the surrounding area and detected obstacles as well as identify other cars. The GPS assists the car in high-level localisation. The camera uses edge detection algorithms to identify and locate the road and the lane markings. All this information together allows the system to make a "conscious" decision [6][7][8].

In more recent days Cho et al also implemented a similar system with a focus on detecting moving objects [9]. There are also cases where neural networks are used for a more successful vehicle and lane detection [10]. As of today the meditated approach seems to be the state of the art in the field as well as the most commonly used. However the end-to-end systems using machine learning has still shown their possibilities of performing well. They promise a more adaptive system architecture in which scenarios does not have to be interpreted and pre-programmed but simply trained and given a correct course of action.

Success definition

When doing a comparison there is need for a clear definition of what is a truth (T), a false positive (FP) and a false truth (FT). Chen *et al* compares their simulated vehicle to a vehicle defined with the ground truth and uses deviation from its driving as a metric. Whilst functional for comparison in this situation it does however not take into account that there are multiple correct ways to drive a vehicle along a road [5].

Koutnik *et al* compares algorithms on the time they take to complete a set track. This definition leaves room for the algorithms to do different choices and only looks at the end result. Speed does however not have to directly relate to successfulness or safety [11].

LeCun *et al* runs the algorithm through a test set of images which will be very measurable but it does not represent that in reality sequence of images given, neither does it take in to account the feedback loop that a real system has. Additionally this method also suffers from the fact that there are multiple ways to correctly drive a vehicle [12].

In NVIDIA corporation's paper they used a naïve approach to simply compare the time a human had to intervene with the total time driven. Whilst difficult to get precise readings from such a method it does take all relevant factors into account and looks at the end result which is reliability [4].

2.2 Artificial Neural Networks

Artificial Neural Networks (ANN) are a type of Machine Learning which seeks to mimic the way our own brains work. The idea of modelling the neurons in the brain dates back to the 1940s [13]. They are in mathematical terms nonlinear statistical models. The general model of a neural network can be seen in Figure 2.1. Generally in machine learning, the two main problems with a model can be split into regression and classification problems. Regression problems seeks to predict vector of real numbers whilst a classification problem seek to give a binary classification to a pre-defined set of classes. The latter is the problem in the popular Imagenet classification challenge [14].

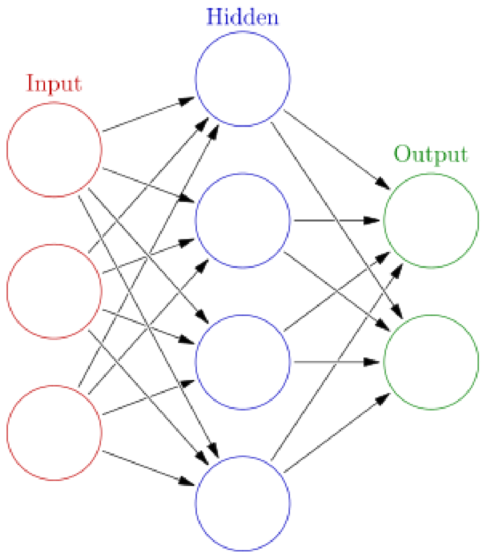


Figure 2.1: The general structure of a Neural Network. This one has only a single hidden layer.

A neural network contains a multitude of unknown weights which decides what the output will be. Modifying these is called the training process. There are many different ways on how to initialise the weights, usually by random. The training process is then usually done through back-propagation which contains two steps. First in the forward pass the network is given an input and calculates an output. An loss function is then calculated from the correct output and the algorithm propagates backwards through the model to correct the weights based on the loss.

There are different ways of calculating the loss function. One of the most common is the mean squared error loss seen in Equation 2.1 where X is the input vector and Y is the n-sized output vector and n is the number of predictions [15]

$$MSE = \frac{(Y - f(X))^2}{n} \tag{2.1}$$

Another loss function used in categorical learning is the cross-entropy loss function which returns the cross entropy between the predicted distribution and the true distribution. Every element in the distributions are between 0 and 1. It can be seen in Equation 2.2 where x is every element in the distribution, p is the true distribution and q is the predicted distribution [16].

$$H(p, q) = -\sum_{x} p(x)log(q(x)) \tag{2.2}$$

The training process is repeated over and over towards a global minimum. However reaching the global minimum is not the ideal case. ANNs often suffer from overfitting as the networks are over sized and contain many more weights than what is actually required of the problem. Thus the global minimum will be the optimal solution for the training data supplied, which is not necessarily the optimal solution for all data. Therefore there are many different solutions to generalise the training.

A simple example is the early stopping where the training is stopped prematurely to avoid reaching the global minimum.

It is possible to either train networks sample per sample or to train in larger batches. When trained in batches the loss functions are summed and at the end of the batch and the model is updated on these losses. This speeds up the training progress [15].

Dropout

Dropout has proved to reduce the overfitting of a neural network. An overfitted network is too adapted to the training data that it would perform poorly on new, unseen data. Dropout works by randomly dropping some nodes in the network during training forcing the network to behave differently each training and therefore develop different, more generic, ways to classify the input [17]. Dropout is only used during training and disabled during testing and usage. One of the drawbacks of using dropout is that the training times is increased by 2-3 times according to Srivastava *et al* due to the updating of the weights are noisy. The noise comes from practically training a different network architecture, due to the dropout, every time, thus taking longer to converge [18].

Activation Functions

Each neural network layer has an activation function which is applied before the output. The activation function for an ordinary fully connected layer takes the dot product of the input matrix and the layer's weight matrix, added with an optional bias, as an input as can be seen in Equation 2.3. Here X is the input, W the weight matrix of the layer, b is the bias and σ the activation function.

$$\Phi = \sigma(W \bullet X + b) \tag{2.3}$$

A traditional activation function inspired by how neurons function is the Heaviside step function which returns 1 if the input is positive and 0 if it is not. Today there are many different kinds of activation functions which will create different results of the network. There are many types of activation functions, how they behave can be seen in Figure 2.2. These differ in aspects of computational speed, upper or lower range which the function may output and many other ways.

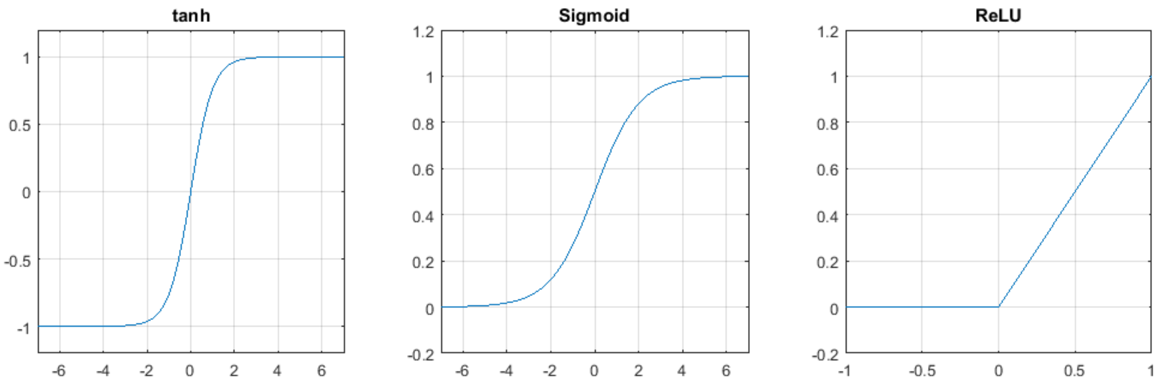


Figure 2.2: Three different activation functions

As of 2015 the most common activation function in deep neural network was the ReLU function (rectified linear unit) [19] which is a non-linear function that returns $f(z) = max(z, 0)$. This activation function was proposed as late as 2011 and was experimentally proven to be an effective method for multi-layer neural networks [20].

Another activation function mainly used in classification models is the Softmax activation function seen in Equation 2.4. This function takes an input vector and reduce them into a real value in the range (0,1). This is useful in classification problems in which the output also is in the range (0,1) [15].

$$\sigma(z)_j = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}} \tag{2.4}$$

Convolutional Neural Network

K. Fukushima *et al* implemented a convolutional neural network in the 1983 called Neocognitron [21]. It was capable of recognising handwritten numerals in different shapes and concluded that it was possible for the neural network to learn other patterns such as letters and geometrical shapes. It was however in 1998 when LeCun came up with his convolutional neural network called LeNet-5 which used both back-propagation and gradient-based learning that CNN showed its true potential [22]. It is thanks to the back-propagation that neural networks are capable of learning complicated multidimensional patterns from training sets with labeled data [23]. Convolutional neural networks was however in the shadow of Support Vector machines which proved to be more successful at classification at the time. It was first in 2012 when Krizhevsky *et al* managed to score a significantly higher score in the ImageNet Large Scale Visual Recognition Challenge using a deep convolutional neural network that CNNs stepped out of the SVMs shadow. They managed to implement a very efficient convolutional operation on a GPU which sped up training significantly [24].

A convolutional neural network could have many different architecture but a simple one could have the following architecture:

Input $\rightarrow$ convolutional layer $\rightarrow$ ReLU layer $\rightarrow$ Max-pooling layer $\rightarrow$ Fully connected layer

The convolutional layer sweeps over the input image with a set amount of filters and stride length creating feature maps. Each filter corresponds to a feature map. The Rectified Linear Unit (ReLU) layer introduces non-linearity to the convolutional neural network. It replaces all the negative pixel values with zero. The max-pooling layer will down sample the output from the previous layer. The fully connected layer will compute the output scores and its size depends on the number of output classes of the network. A fully connected layer means that each neuron in this layer is connected to every neuron in the previous layer.

The convolutional layers purpose is to extract features from the input by using filters, also sometimes called kernel or feature detector. The filter is a matrix with a fixed size and slides over the image with a stride length. At each position it computes the dot product creating a feature map. The neural networks learns how the filters should look during training and are therefore initialised randomly.

A layer often used in CNNs is a max-pooling layer. The purpose of the max-pooling layer is to reduce the dimensionality of the input and therefore reduce the number of parameters to learn and to also achieve spatial invariance [25]. The max-pooling layer works by taking the maximum value in a n-by-n region that strides over the previous layers output with a fixed stride length and thus reducing the dimensions of the input but keeping the most relevant data. The most common size of the region is 2x2 with a stride of 2 which creates a non-overlapping pooling. Krizhevsky used however a 3x3 region with a stride of 2 which created an overlapping pooling which increased their classification performance [24]. For example using a 2x2 region with a stride of 2 would decrease a 28x28 input to 14x14 output.

Scherer *et al* did however an empirical evaluation of different pooling and stride sizes claiming that overlapping pooling regions has no significant improvement over non-overlapping regions [25].

Recurrent neural network

Recurrent neural networks has become the choice of neural networks when it comes to handling sequential data. Examples of these includes text, sound and video. RNNs has proven useful for handwriting recognition, speech recognition, language modelling and video classification. Graves et al [26] used RNN with a Bidirectional Long Short Term Memory (BLSTM) to recognise handwriting and argued it to be superior to the then state-of-the-art Hidden Markov Model-based systems. Mohamed et al [27] tackled speech recognition using RNN with Long Short Term Memory (LSTM) and argues they achieve the best recorded result on the dataset. Mikolov et al [28] improved an RNN language model. What all these issues have in common is that a single datapoint does not hold enough information to define the data.

The basic algorithm of the RNN is much like a normal neural network layer but it has a separate set of weights to evaluate the result of the previous element in the sequence's output. There are many different variations of it, but the one used by the framework Keras used in this thesis is the Elman network. This hold a hidden state calculated by Equation 2.5 where x_t is the input and $h_t$is the hidden state. W_h and U_h is the correspondent weight matrices. b_h is a bias. σ is the sigmoid activation function.

The hidden state is then used to calculate the output in Equation 2.6 where y_t is the output for time t, h_t is the hidden state from Equation 2.5, W_y is the weight matrix for the output and b_y is a bias. [29].

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h) \tag{2.5}$$

$$y_t = \sigma_y(W_y h_t + b_y) \tag{2.6}$$

Long Short-Term Memory

LSTM was proposed 1997 by Hochreiter and Schmidhuber. RNN suffers from having issues storing information over a longer time period as "... the errors flowing backwards through time tend to (1) blow up or (2) vanish". This creates issues when the sequences are very long. It is still possible for the RNN to learn but requires extensive training. LSTM was the solution to this which has units which can retain the errors through time [30]. The main downside however is the increased computational complexity that comes with the memory component. The performance of the LSTM is often at least equal to the RNN and will be superior as the sequence length increases. A further step is the BLSTM (bi-directional LSTM) where every memory module has a connection backwards as well as forward. Experiments on sequential data by A. Graves suggests that the BLSTM is the most successful algorithm followed by the LSTM and then the RNN. He also notes that the LSTM and BLSTM has a tendency to overfit on extensive epochs [31].

2.3 Inertial Measurement Unit

An Inertial Measurement Unit (IMU) is a device used for tracking the movement of an object. It uses inertial sensors such as gyroscopes and accelerometers. Usually three mutually orthogonal accelerometers and three gyroscopes, each one aligned for one of the axis's of the accelerometer, are used. The accelerometer measures the acceleration in each axis and the gyroscope measures the angular velocity in each axis. Accelerometers are often used in the automotive industry where typcial applications are active safety systems such as airbags and electronic suspensions [32]. Combining the accelerometer with a gyroscope, thus creating an IMU, makes it possible to use the information for navigation purposes. This is for example used in video-camera stabilisation and gamepad controller but also quadrocopters to ensure its orientation [33].

The IMU has error sources that has to be dealt with. These errors are dealt with by calibrating it. One error source that has to be dealt with is temperature. Say for example that the gyroscope is calibrated at a certain temperature and if the temperature differs during usage it will lead to an orientation error that will grow linearly with time. One way to handle this is to calibrate the gyroscope at multiple temperatures and include an temperature sensor in the IMU. Both accelerometers and gyroscopes have bias errors. For accelerometers it is measured in milli-g or micro-g and for gyroscope it is often measured in deg/hr, or in lower grade gyroscopes in deg/s. If the gyroscope bias is not accounted for the error will increase over time when integrating the angular velocity to degrees. Some bias errors are static and can be compensated for during calibration. There are however dynamic bias that vary from run to run and are therefore hard to compensate for. The dynamic bias is around ten percent of the static bias [34]. There are many other error sources according to A. D King including scale factors, instrument misalignment, non-linearites [35].

Chapter 3

Methodology

3.1 Neural Networks Development Method

Designing a neural network is not straight forward. There is a very large amount of different variables that may be modified. The variables consists of not only the structure and parameters of the network itself but also in how much training data should be used and how this data should be pre-processed. Tackling this issue with a trial and error approach is time-consuming as each train and test step may take seconds to minutes to hours and with the amount of variables available even seconds is a long time.

Train, Cross-validation and Test Data

Using a single training sample for intensive training will yield a very low training loss, as the model will learn to fit that training data perfectly. However, that single sample does not present the whole dataset and the model will therefore perform very poorly in reality. That is why the complete dataset is split into training and test data. After the model has been trained on the training data its performance is evaluated on the test data, which for the model is completely new data it has never seen before.

One issue with only splitting into training and test is that the model will become optimised towards the test data - which may not necessarily represent reality perfectly. Therefore the cross-validation set is introduced as a middle step. The data is split within the three parts of test, cross-validation and training. The model is trained on the training data, tested on the cross-validation and optimised towards the cross-validation. The model is finally tested on the test data to ensure that the model can produce a generic result. It is important not to use the final test data for tuning and only for evaluation[15].

In this project the data will be split into training and test only. This is since the model will also be tested on reality which will prove its final performance.

Variance and Bias

The problems the current network iteration faces can be categorised into two general cases - bias or variance issues. Bias is when the network model is too generic for the problem. This can be seen if the training and test errors has converged to the same value, but the converged error is still too high.

Variance - or overfitting - problems are when the network model is too complex and/or with the sufficient amount of training data. These problems are characterised by a low training error and with a high test error. The model has simply learned the training data so well it cannot generalise.

Bias problems can be solved by making the model more complex, allowing it to better fit the data. They can also be solved by making the data simpler with less features.

Variance problems on the other hand can be solved by simplifying the model, restricting the possibilities to overfit. Another solution is to increase the amount of training data [15].

Learning Curve

To clearly see the difference between a bias and variance issue a learning curve can be generated for each network. An example of this can be seen in Figure 3.1. Here the network is trained on a small subset of the total training data and then tested on the full test data. The training and test loss are then added as data points in the figure. The model is then retrained on an increasing larger subset of the training data to the point where all available training data is used. It can then be possible to see the difference between the training and test loss. If this difference is small then there is a bias problem. If this difference is large then it is a variance problem. If the derivative between the few last test losses is strongly negative then it is possible to extrapolate that more training data will improve the loss [15].

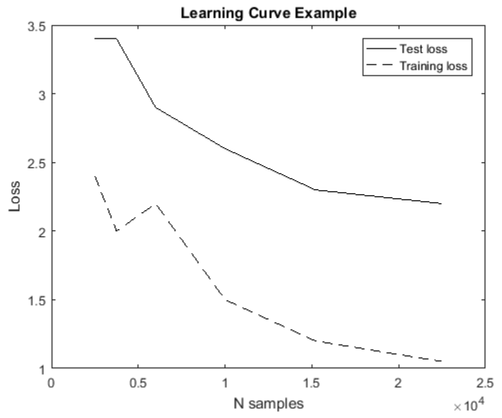


Figure 3.1: Example learning curve. The x axis is the number of training samples and y axis is the result of the loss function. This specific graph shows variance issues and that more training data may continue to improve the result.

3.2 Keras and Theano

This project uses Keras 1.2.2 with Theano 0.9.0 as backend.

Keras is a high-level neural networks API written and used in Python and uses either Theano or Tensorflow as backend.[36] Theano is a "Python library that allows to define, optimise and evaluate mathematical expressions using multi-dimensional arrays efficiently". It is an open source project and is widely used in machine learning. [16]

The Keras API allows for a high-level and practical approach to neural network development. It is quick to get started and there is not a direct requirement of heavy theory.

A network is quickly modeled layer by layer, compiled and then fitted. Example code can be seen as:

```
#   Create a model of the sequential type
model = Sequential()
#   A fully connected layer with a linear activation function.
#   The input dimension is defined in the first layer
model.add(Dense(128, activation = "linear", input_dimension=(1,10)))
#   Hidden layer with a "relu" activation function.
#   The final layer defines the dimension of the output
model.add(Dense(1, activation = "relu"))
#   Compile the model to verify that it is a functional model and
#   provide the over-arching settings of loss function and optimiser.
model.compile(loss = "mean_squared_error", optimizer = "sgd")
#   Fit the model to pre-defined training data where
#   x contains the input and y contains the correct output.
#   Multiple options such as how many epochs and the batch size
#   can also be chosen.
model.fit(train_x, train_y, epochs = 5, batch_size = 32)
#   Evaluate the trained model on pre-defined test data
#   Returns the loss and other metrics defined.
evaluation_loss = model.evaluate(test_x, test_y)
```

This piece of code makes a network where the input is an array of 10 numbers and the output is a single number. It has two fully connected hidden layers and uses Mean Squared Error as its loss function. The optimiser is stochastic gradient descent, which is the mathematical optimisation algorithm used.

Keras Functionality

Here is a short explanation of some Keras functions used in this thesis.

Dense: A Dense layer is a regular densely-connected Neural Network layer. It takes in the input array and makes a dot product with the layer's weight matrix, optionally adding a bias and then sending the result through an activation function. If no activation function is manually chosen a linear function is assumed.

Conv2D: A convolutional layer which takes the input matrix convolves this with a convolution matrix, or kernel, and optionally applies a bias and/or an activation function to the output. It is possible to decide the size of the convolution kernel as well as its stride length.

MaxPooling2D: A pooling layer which takes an input matrix and performs max pooling before sending it to the output. It is possible to decide the size of the pooling.

LSTM or SimpleRNN: A recurrent layer which takes a sequence of vectors and applies the associated algorithm. Output size is determined by the size of the layer. It is possible to choose the activation function for both the recurrent step and the output step.

Wrappers: Keras has a series of wrappers which modifies an ordinary layer to hold specific functionality. These include 'TimeDistributed' which applies a layer to a time distributed sequence which allows for example convolutional layers to be applied to a sequence before sent to an LSTM.

Another wrapper is 'Bidirectional' when applied to a recurrent RNN or LSTM layer transforms it to be bi-directional.

Callback Functions: There are multiple functions which allows Keras to perform actions in the training procedure. Those used in this thesis include 'History' which saves the training loss for each epoch in an object and 'ModelCheckpoint' automatically saves the model after every epoch.

3.3 Multiple State Network

It is an open question on how input a sequence of events into a multi-state network as the RNN. Here the considered options are explained.

Sequence of raw images

The most common ways of preparing a sequential dataset, that will be tested, are:

- Input sequence of raw images.
- Input sequence of pre-processed images.
- Input sequence of pre-processed images with single-state model.
- Time-distributed CNN.

The first naïve method is to simply gather the images in sequences and input them to the RNN without any earlier layers. However when it comes to raw images the amount of parameters is very large ($width \cdot height \cdot channels$). To learn this it would require a very large database of images.

Another option here is to reduce the number of parameters by pre-processing images. Options include converting the image to greyscale or binary through clever algorithms. This will destroy image information that may or may not be important. Additionally it is possible to pre-process all images with a single-state model that has already been taught to steer. To avoid destroying too much information the single-state's signal is intercepted by redirecting a layer that is not the final layer and uses this output as an input to the multiple-state model.

A final method is to apply a time distributed CNN to each image in the sequence. This will teach and apply a CNN layer to each image in the sequence and thus have an adaptive filter to each and every respective element.

3.4 Test Scenarios

Here are the multiple scenarios to train and evaluate three different networks in. The three networks that will be tested includes a traditional single-state model, a 5-state RNN and a 25-state RNN. Scenario 1 is to evaluate base performance of driving along a road. Scenarios 2 and 3 evaluates situations which are believed to depend on sequential information.

Result Evaluation Method

The result of the following scenarios will be analysed through comparing the multiple-state models with the single-state model. The ideal run serves as a reference case as it shows how an 'ideal' drive is, which is a human. Additionally since it was taught by samples from a human, if the network has learned well it should be similar to the ideal drive.

The things that will be analysed from the test results are not only just raw success rate, but also if the multiple-state networks emit any special behaviour when used. Additionally when analysing the results the standard deviation plays an important role. Given a perfect network given more-or-less the same scenario it should give the same outcome. If not, it implies that the network has failed to generalise and will take a lot of different decisions to two sets of data that is meant to mean the same thing. Additionally if a network takes a large amount of faulty decisions this will create outliers in the direct output of the network which are also measurable as deviation from the mean. In short; consistency in combination with good results implies a good network. The physical results will also be compared to the theoretical result as additional confirmation.

Scenario 1: Driving in an eight

Intro: Evaluation of the three methods' general driving capability along a continuous track.

Method: Tape an eight track. Gather data by driving along the track as centralised to the middle as possible along this road. Train the three networks and let them drive along the eight track.

Output:

- Track completion [Yes/No]
- Travel Path [(x, y, t)]
- Steering Request

Discussion: This scenario seeks to evaluate general driving performance. An eight contains all basic driving scenarios such as straight road, right turn and left turn. Additionally it will allow for easy gathering of data as there is no need to stop and reset.

Scenario 2 : Evaluation of a parallel movement

Intro: Consider a car driving in front of the autonomous vehicle. An intuitive thing is to try to keep a constant distance to the car ahead which would require a sense of how fast the leading car goes. If the car goes faster you go faster. If the car slows down you slow down.

Method: Tape two lines creating a straight road. Set up a motor which pulls another 'car' across the road in front of the autonomous vehicle. The pulled 'car' will decelerate and accelerate softly to simulate another driver. The autonomous car shall as well as the human driver keep a constant distance between itself and the leading car.

Output:

- Track completion [Yes/No]
- Distance to leading car [$(\Delta x, t)$]
- Travel Path [(x, y, t)]

Good is to complete the track without collision with the leading car and keeping a constant distance to the leading car on par with the training driver. The expectation is that both networks will be able to follow the car but RNN will have a more natural and 'smooth' movement.

Discussion: This scenario hopes to evaluate the fact that a single image does not tell the direction of movement. The hypothesis is that the multiple-state network will be superior.

It is possible to mount an ultrasonic distance sensor on the front of the car to properly measure the distance between the car and the object ahead. Additionally it can be used to generate a more precise ideal reference run if combined with a PID controller.

Scenario 3 : Evaluation of conflicting decisions

Intro: Consider driving a car along a large road. An object is in the way. You go around it. Either you go around the right of the object, or you go around the left. Both are equally viable. It is however important to choose a direction and keep going around the object that way. This is not something intuitive to a neural network which may take many different contradicting decisions per second.

Method: Tape a slightly wider lane road. Place an object in the middle of the lane. Train the car to go around the object. Half the time on the left side. Half the time on the right side.

Output:

- Track completion [Yes/No]
- Choice taken [$Right/Left/None$]
- Collision [Yes/No]
- Travel Path [(x, y, t)]

Good output is to have a 'determined' motion pattern where the car chooses a path around and sticks to it. The expectation here is that the RNN will be able to keep to a certain pattern which the CNN can not.

Discussion: This scenario hopes to evaluate two things. Firstly the theory that the multiple-state network has the ability to hold on to a longer 'plan' or trajectory and will be able to continue around the object and not 'forget' what it is doing half-way. Secondly it is possible that the multiple-state network will not have issues with mixed decisions as it will trail into a trajectory and continue along it.

This test requires the system to have a fast image update frequency. The reasoning is that for this theory to be tested properly the networks needs multiple quite similar images in a row. This lets the system take many different decisions before the real vehicle has managed to move enough for the image to majorly change.

The data recorded needs to be close to 50/50 in terms of which way around the item is taken to not create a bias.

3.5 System Architecture

The system consists of an RC car and three separate computing nodes; a PC, a Raspberry Pi 3 and an Arduino Mega 2560.

RC car

The RC car is provided by Cybercom. The manufacturer and model of fthe car is unknown but it has a DC-motor with encoder and a servo. The DC-motor shaft is connected to the cars differential drive train making it a four wheel drive vehicle. It has been used in previous master thesis's at Cybercom and it provides a good platform to build upon for this thesis. The car will be equipped with two mounting plates for a battery pack,the Raspberry Pi 3 , the Raspberry Pi Camera Module and the Arduino Mega 2560.

PC

Training and using a neural network is a task which may require extensive computational hardware. They are also greatly sped up by running the calculations on a dedicated graphics card. Therefore a separate PC was chosen to deal with these tasks.

The PC receives a new image bundled with the car state from the Raspberry node. It then inputs these in to the neural network and received a steering prediction. This gets sent back to the Raspberry node. The camera image is displayed on the PC. A controller is connected to the PC to steer or control the velocity of the vehicle if needed. The controller commands are also sent to the Raspberry Pi node.

Raspberry Pi 3

The Raspberry Pi 3 is an affordable single chip computer with a GPU, multiple GPIO ports and a dedicated CSI (Camera Serial Interface). This is used together with a Raspberry Camera Module connected to the CSI.

Its core usage is as the main communication node. The camera continuously captures image frames. The Arduino continuously sends the car state. The Raspberry communicates these at regular intervals up to the PC and then waits for a steering command. When this arrives it communicates it down to the Arduino.

Arduino Mega 2560

The Arduino is an open source microcontroller board made for rapid prototyping and learning. It was chosen as it is very 'plug-and-play' and user friendly.

The main task for the Arduino is to read and control the hardware components. Three components are connected to the Arduino. The first is a motor with attached encoder. The motor is controlled through an h-bridge, the SparkFun Ludus Protoshield, with a PWM signal. The encoder pulses are read and interpreted with interrupts. A servo for steering controlled with a PWM signal. The 9-axis IMU which gathers data from a 3-axis gyroscope, a 3-axis accelerometer and a 3-axis magnetometer.

At periodic times the Arduino sends the current velocity, steering and IMU data up to the Raspberry. When the Arduino recieves a steering command this is converted to a PWM and sent to the servo. When the Arduino recieves a velocity command this updates the reference value for a PID-controller which continously updates the speed.

Chapter 4

Implementation

4.1 Mechanical Design

RC-Car

The RC-car that was given by Cybercom was equipped with a DC-motor with a quadrature encoder and a servo. Usually a modern basic servo has three wires; one for power, one for ground and one for PWM signal which controls the wanted position of the servo. The servo of the car had five wires as it lacked a dedicated controller module; two for the motor and three for the potentiometer. Whilst it required manual controller software this eventually proved to be an advantage as it was possible to get direct position feedback from the potentiometer. This is normally not possible from a simple servo as it takes position commands but does not communicate where it is. Additionally the resolution of the servo reaches $0.18°$/step limited by the 10-bit resolution of the Analog-to-Digital converter in the Arduino. Finally it allows the user to modify the controller code to slow down the reaction time of each command which reduces the effect of single faulty decisions of the network.

The RC-car did not initially have any surface to mount the needed components on so a mounting plate had to be designed. This design can be seen in Figure 4.1. It was not practical to everything on a single level so two levels had to be designed with spacers in between. Holes were added to mount the Raspberry Pi, the Arduino and the camera mount on the top layer. The power bank is placed on the lower level and is fastened using Velcro. To avoid unnecessary heating due to closed space and to allow cable passage slits were added wherever there was free space.

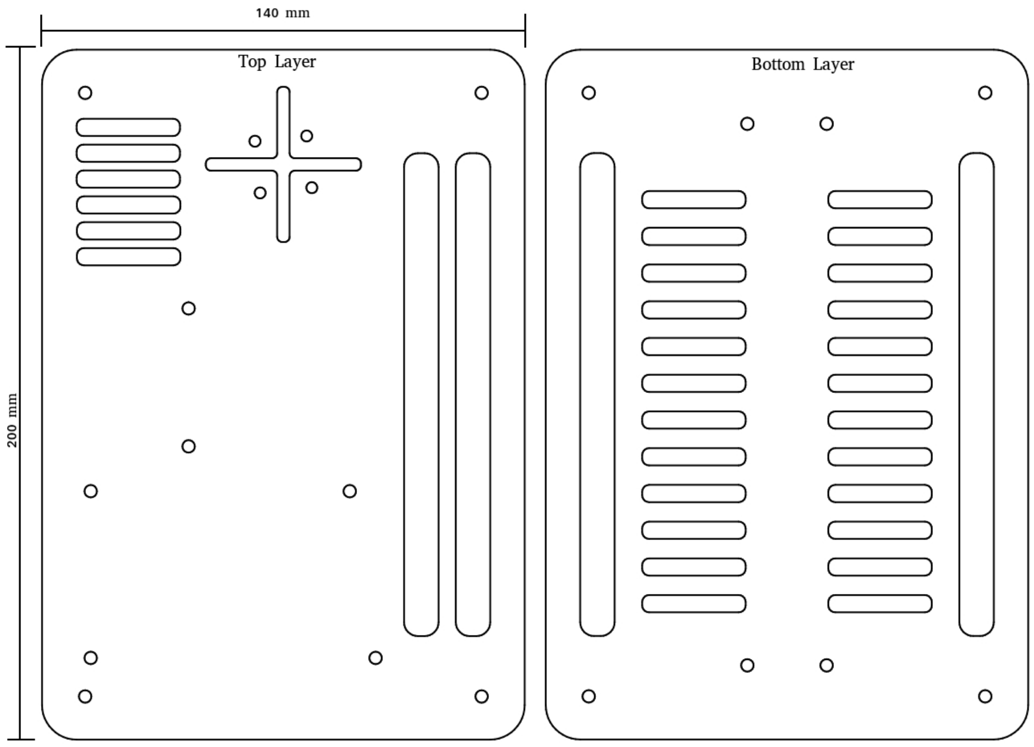


Figure 4.1: Mounting plate designs.

A mount for a HC-SR04, an ultrasonic sensor, was made in CAD and 3D-printed. It was mounted in front of the car and used for checking the distance of the object for scenario 2. The final construction of the vehicle can be seen in Figure 4.2. In the back of the vehicle one can see the Arduino Mega powered by a battery on the lowest level and connected via USB to the Raspberry Pi 3. The vertical component mounted on the Arduino is the IMU.

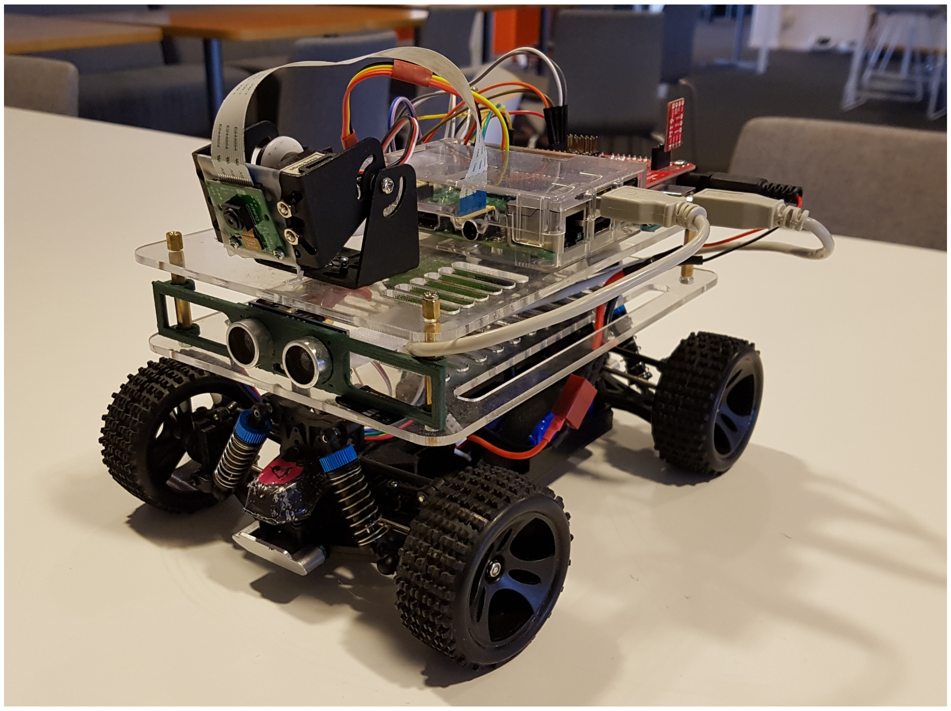


Figure 4.2: The final construction

4.2 Robotic Operating System

Robotic Operating system (ROS) is used for communication between the different computation devices [37]. ROS Kinetic was used in this project. Each program in a ROS network is called a node and all these nodes are connected to a core, called roscore, running on one of the computers. By just telling each computation device the IP of the computer running the core ROS handles the communication between all the nodes regardless where the node is running by sending data over Wi-Fi. A ROS program can be written in Python, C++ and Lisp, and by using Matlabs Robotics System Toolbox in Matlab and Simulink. A basic ROS setup could be that node **A** publishes a message to the topic *Speed* containing information of the robots current speed. Node **B** has subscribed to the topic called *Speed* and each time node **A** publishes a message to the *Speed* topic a callback function will be called in node **B**. This callback function could for example save the speed data to a text file.

A built-in function in ROS is the ability to record every message published, containing the message data and time-stamp, in something called a rosbag. Rosbag is a command-line tool where you can either choose which topics you want to record or simply record data from every topic. A rosbag, set to record every topic, was created for each scenario test run to store all the data for evaluation. Matlabs Robotics System Toolbox has support for reading data from rosbags and was therefore used to evaluate the data in Matlab.

Serial Communication

ROS has created a library for Arduino. This makes it possible to create nodes, subscribing and publishing messages on an Arduino using serial communication. An Arduino Mega 2560 is connected to a Raspberry Pi 3 with a USB-cable which was only used for serial communication, the power source for the Arduino came from a 7.2 V battery connected to the Arduino power jack. The ROS package used for serial communication was rosserial_arduino, [38]. The default baud-rate for the ROS serial communication is 57600 but was changed to 115200 due to the amount of data needed to be sent to and from the Arduino continuously. The baud rate was changed thanks to that the ROS serial node warns when a message has be lost due to overfilled buffer and increasing the baud rate solved this problem. Messages to the Arduino include requested steering and requested motor speed, and messages from the Arduino includes vehicle speed, gyroscope data and accelerometer data.

Motor Control

The motor, encoder and gearbox data were initially unknown due to lack of model numbers or markings. The encoders pulses per revolution were estimated by rotating the wheel one revolution and counting the number of pulses by connecting channel A on the encoder to an interrupt pin on the Arduino. This number was estimated to 300 per revolution. Using the circumference of the wheel and a fixed sampling time the velocity was calculated, the equation can be seen in Equation 4.1 where ΔP is the number of pulses since last measurement, r is the wheel radius, P_r is the number of pulses per revolution and v is the vehicle speed. To evaluate if the speed was correctly calculated the speed was integrated over a fixed sample time to get the driven distance. A distance was then measured on the floor comparing the actual driving distance with the calculated distance. The accuracy was ± 2 centimetres per meter driven which deemed accurate enough for the project.

$$v = r \cdot \frac{\Delta P \cdot 60}{P_r} \cdot \frac{2 \cdot \pi}{60} \tag{4.1}$$

Initially the velocity commands ranged from 0 to 255 which was directly applied as PWM signal to the H-bridge. This created issues whenever the car's conditions changed, mainly when weight was added or removed, that a PWM no longer represented the same velocity. As the networks uses velocity as a teaching variable the given speed needs to actually be that speed. Therefore a PID-controller was implemented so the velocity commands could be actual velocity. Every new command updates the reference value in the controller which modifies the PWM signal to achieve this speed. The velocity commands comes from a topic called motor_request the Arduino subscribes to. The callback function of the motor_request message simply updates the reference value in the controller. The PID controller is updated using one of the timers in the Arduino Mega to make sure it updates at a fixed rate which is set to 10 Hz. The top speed of the vehicle is 1.2 metres per second when the DC motor is connected to a 7.4 V lithium polymer (Li-Po) battery.

IMU

A MPU-9250 - SEN-13762 Breakout from SparkFun was used as IMU for the car. It is a 9 Degrees of Freedom IMU including a 3-axis accelerometer, 3-axis gyroscope and a 3-axis magnetometer. Sparkfuns own arduino library was used for the MPU-9250 [39]. The sampling frequency for the IMU is set to 20 Hz. The IMU data will be used together with the vehicle speed to be able to create an odometry message containing information of the vehicles local coordinates and rotation to then plot the driving path in Matlab for evaluation.

The angular rate sensor on the MPU-9250 can be user-programmable to different settings depending on the accuracy needed. These values are ±250, ±500, ±1000 and ±2000 °/sec. The chosen rate was ±250 since we deemed it accurate enough for its application. By multiplying the raw gyro data with the chosen rate the angular velocity can be calculated.

The accelerometers full scale range on the MPU-9250 can also be changed. The range is the limit of the acceleration which the accelerometer can measure. The lower the range a more precise measurement can be achieved. The values are ±2, ±4, ±8 and ±16 g. The chosen range was ±4 as the top speed of the vehicle will be 0.4 m/s and will very unlikely reach more than 4g. The vehicle might reach those values when it starts from standing still but what the accelerometer was meant to check the acceleration when turning and those forces are much lower than a start from standing still.

The IMU is mounted on the Arduino and reads the IMU data at 20 Hz. The Arduino publishes a ROS message with the angular velocity, deg/s, and not the raw gyro data. This is done multiplying the raw gyro data with the chosen rate. The gyroscope, accelerometer and magnetometer data are each stored in a separate ROS Vector3 message which is a message type containing x,y,z components. The Arduino was set to publish the gyroscope and accelerometer messages with a frequency of 10 Hz because a higher publishing rate resulted in many package losses. The magnetometer data was disregarded because we found no reason to use it.

Odometry

By combining the vehicle speed and the angular velocity from the gyroscope on the IMU the position and orientation, so called pose, of the vehicle relative to the starting position can be calculated. This method for calculating position suffers however to errors from integrating over time and also the drifting from the gyroscope. These errors can in this case however be neglected due to the short distance and time each test takes.

ROS handles odometry by creating a so called odometry-message by using multiple sensor inputs. In this case the translational speed of the vehicle was calculated from the encoder on the DC-motor and the angular velocity in the horizontal plane measured from the gryoscope on the IMU connected on the Arduino. For ROS to interpret the pose of the vehicle a global Cartesian coordinate system, called frame in ROS, and a local Cartesian coordinate system for the vehicle are set up. The global frame, called map, is a stationary coordinate system while the local one, called base-link, is positioned in the middle of the vehicle. The forward direction of the vehicle is aligned with the X-axis of the base-link frame which means that where the X-axis is pointing so is the front of the vehicle. The base-link frame moves relative to the map frame and can thus be used to calculate position relative to starting position and easily be visualised using ROS built in visualisation tool Rviz but can also be used to plot the path in Matlab for evaluation of the driving performance.

For each iteration of the odometry program the new position of the vehicle is calculated. The first step is to calculate the difference in the X,Y and Yaw, θ, of the vehicle using Equation 4.2 where v is velocity of the vehicle, v_θ is the angular velocity and Δt is the time since the last iteration.

$$\begin{aligned} \Delta X &= v \cdot \cos\theta \cdot \Delta t \\ \Delta Y &= v \cdot \sin\theta \cdot \Delta t \\ \Delta\theta &= v_\theta \cdot \Delta t \end{aligned} \tag{4.2}$$

These values are then added to the vehicles previous relative position to the map frame. A Quaternion message is then created from the Yaw using a built-in function in ROS called tf::createQuaternionMsgFromYaw().

Using the updated values of the vehicle an odometry message is created containing the current time, the vehicles relative X and Y position and the quaternion message for vehicle orientation. The message is then published. The rate of the odometry node is set to 10 Hz.

Camera

A Raspberry Pi Camera Module V1.3 was used, connected to the CSI port on the Raspberry Pi 3. The ROS camera node was written in Python using the PiCamera package as interface for the camera. The camera was set to capture images in a size of 320x240 pixels. The capture rate was set to 25 frames per second. The raw image was compressed to jpg to reduce the bandwidth needed to transfer the image. The bandwidth usage of the raw image was 5.80 MB/s and the bandwidth usage for a compressed image was 1 MB/s when capturing images at 25 frames per second. We managed to increase the capturing rate to 80 frames per second which would increase the bandwidth usage to 20.90 MB/s for the raw image and 3.50 MB/s for the compressed image but since the neural network would run at 20 Hz we decided to settle for a capture rate of 25 frames per second to avoid unnecessary processing load on the Raspberry Pi and also due to the limitations of the WiFi used. The Raspberry Pi 3 has a built-in 802.11n BCM43438 WiFi module [40]. The transfer speed was tested between the Raspberry Pi and the laptop running the neural network and we got an average transfer speed of 3.4 MB/s using iPerf, a tool for measuring maximum achievable bandwidth on a network. This shows that if we were to use a higher frame rate or uncompressed images we would maximise the bandwidth with only image data and that would not be feasible due to other messages needs to be sent as well.

Joystick

In order to gather training data it was necessary to control the vehicle in a smooth manner. The solution was to connect a Playstation 3 Wireless Controller via Bluetooth to the laptop. By combining ROS and Pygame, a Python library, it was possible to read the values of the controllers joysticks and button and publish these as steering and motor request messages. The joystick node reads controller values and publishes the values with a frequency of 10 Hz which deemed fast enough for a rapid response. During the data gathering phase of the project a button was set to start recording data when held down which streamlined the data gathering process. This was also done for the recording of the test results to make sure each run started on equal terms.

4.3 System Architecture

The hardware architecture of the vehicle can be seen in Figure 4.3. It shows how each component on the vehicle is connected and which direction the communication/data comes from. The H-bridge is a Toshiba TB6612FNG which is integrated on a Ludus Protoshield capable of controlling two separate motors. The shield is then mounted on the Arduino Mega. The power source for the Raspberry Pi is a small power bank which outputs 5 V DC 2.1 A and the power source for the Arduino Mega is a 7.4 V 1100 mAh Li-Po Battery. The ultrasonic sensor was connected to the Raspberry Pi instead of the Arduino Mega to avoid increasing the load of the serial bus between the Raspberry Pi and Arduino and risk losing messages.

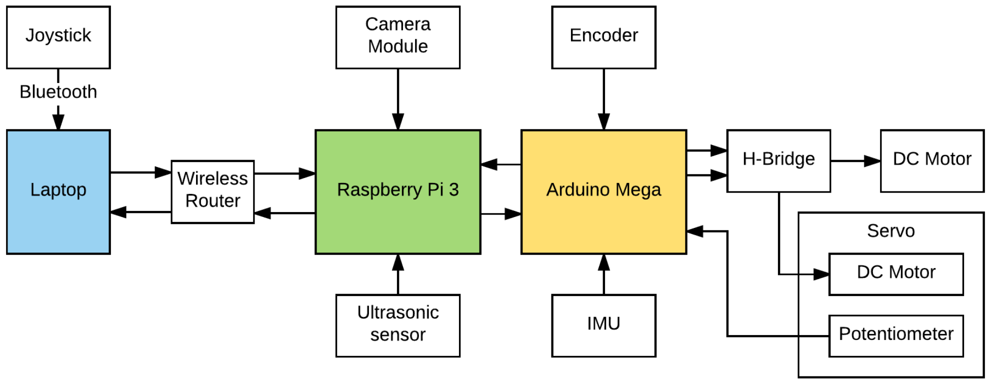


Figure 4.3: The hardware architecture of the vehicle

ROS nodes are running on each of these computing units with the roscore running on the laptop. The ROS network architecture can be seen in Figure 4.4. The background color of each box corresponds to which computing unit the node is running on, but the unit it also written in the box. When an arrow is pointing towards a node it means that that certain node has subscribed to a message published by the origin of the arrow. The arrows which are pointing out to nothing, as for example the distance node, shows that the node publishes data but no node is subscribing to it. The distance publishes a distance to the object in front of the vehicle and this is only relevant when evaluating scenario 3 and has therefore no subscriber. When recording a rosbag it will record all the published messages. The reason both the Joystick node and the Neural network node publishes motor request and steering is because it depends on which scenario that is run. When the network only controls steering the joystick will handle the motor request and vice versa. A computation cycle of the network starts with a captured image from the camera on the Raspberry Pi sent over Wi-Fi via a router to the neural network running on the laptop. The network predicts a steering angle or motor request speed and sends it to the Raspberry Pi which sends it via serial communication to the Arduino which actuates the servo or the DC-motor.

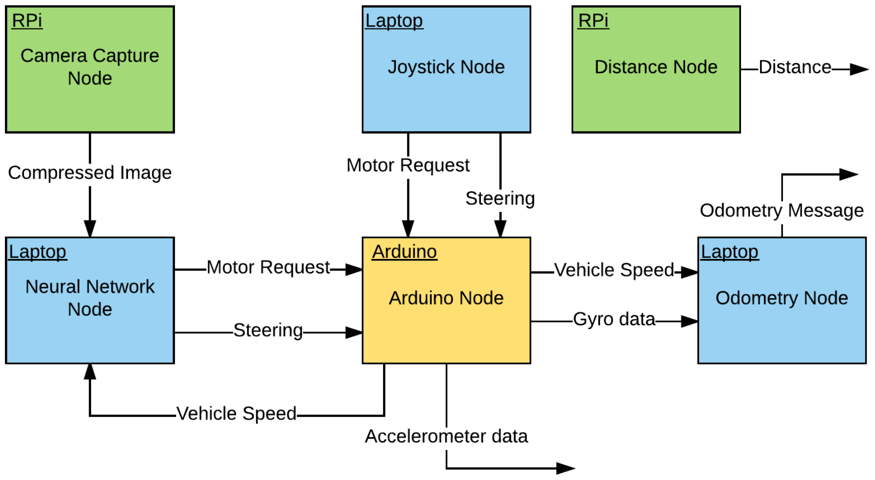


Figure 4.4: ROS network architecture

4.4 Neural Networks

The networks were implemented in Keras using Theano as backend. The training was done on a Nvidia GeForce GTX 1080, an Intel(R) Core(TM) i7-6700K CPU and with 32 GB of RAM.

The general training process was to start with a simple model that converged when trained on the full dataset. A learning curve then became generated by training the model on an increasing subset of data and evaluating every training step on the test data. At the end of every learning curve it could be seen if the model is too complex, if it was too simple and if more training data will help.

Complexity in a model can be interpreted as the size of the model, or how many possible parameters exists. The more parameters a model has the more complex shapes can be learnt by it. The amount of parameters can be increased by increasing the number of hidden layers, the number of neurons in the layers as well as special parameters in the more complex RNN and CNN layers.

A new network was created for each scenario. Two general steering models, single state and multiple state, were created for Scenario 1 and 2, two velocity models for scenario 2 and two steering models to avoid an obstacle were trained for scenario 3.

General Model

It was important to design networks that were relatively similar and had comparable results. An initial question was whether to use regression or classification. Here regression has a single output which is the true value whilst a classification model has an output for each class which contains the likelihood of that specific class. Initially regression was chosen for both steering an velocity as the goal was to predict a single number. However in the case of steering where the value could be defined to 100 classes (0-100) proved to be equally effective but now contained more information for debugging as it was possible to see the full spectrum of the model's decisions.

One could argue that regression scores a more accurate loss function in cases where the model is slightly off the correct answer whilst the classification model only gives a correct if the model is spot on the correct classification. Then a negative part is that if the network can not decide if it is full left or full right regression will output the average which is straight forward. Classification would simply decide on the most likely of the two and thus 'maybes' will not modify the end answer.

Velocity is defined as a floating point value and did not have these conflicting answers and therefore used regression.

Data gathering

For the scenario 1 general steering model the car was driven along an eight track seen in Figure 4.5. The eight track was chosen as it contains all relevant steering situations such as left turn, right turn and straight road. It also allowed the data gatherer to drive indefinitely without any need to reset the car. Images were gathered at a rate of 20Hz and was matched with a set of data which included the gyroscope, requested velocity, requested steering, actual vehicle speed and accelerometer. For scenario 1 a total of over 77 thousand images were gathered, which corresponds to roughly one hour of driving. The cardboard boxes around the track was removed for half of data gathering to have to variations of the surrounding.

For the scenario 2 velocity model a construction was required to pull an object in a repeatable manner. Therefore a spool was 3D-printed and connected to a motor. This construction can be seen in Figure 4.6. The motor was controlled using an Arduino UNO and a NPN-transistor. The Arduino generated a sine wave signal with randomised frequencies and amplitudes. The driver had to attempt to keep a constant distance to the object. For scenario 2 over 16 thousand images were recorded this way, which corresponds to 77 different data gathering runs.

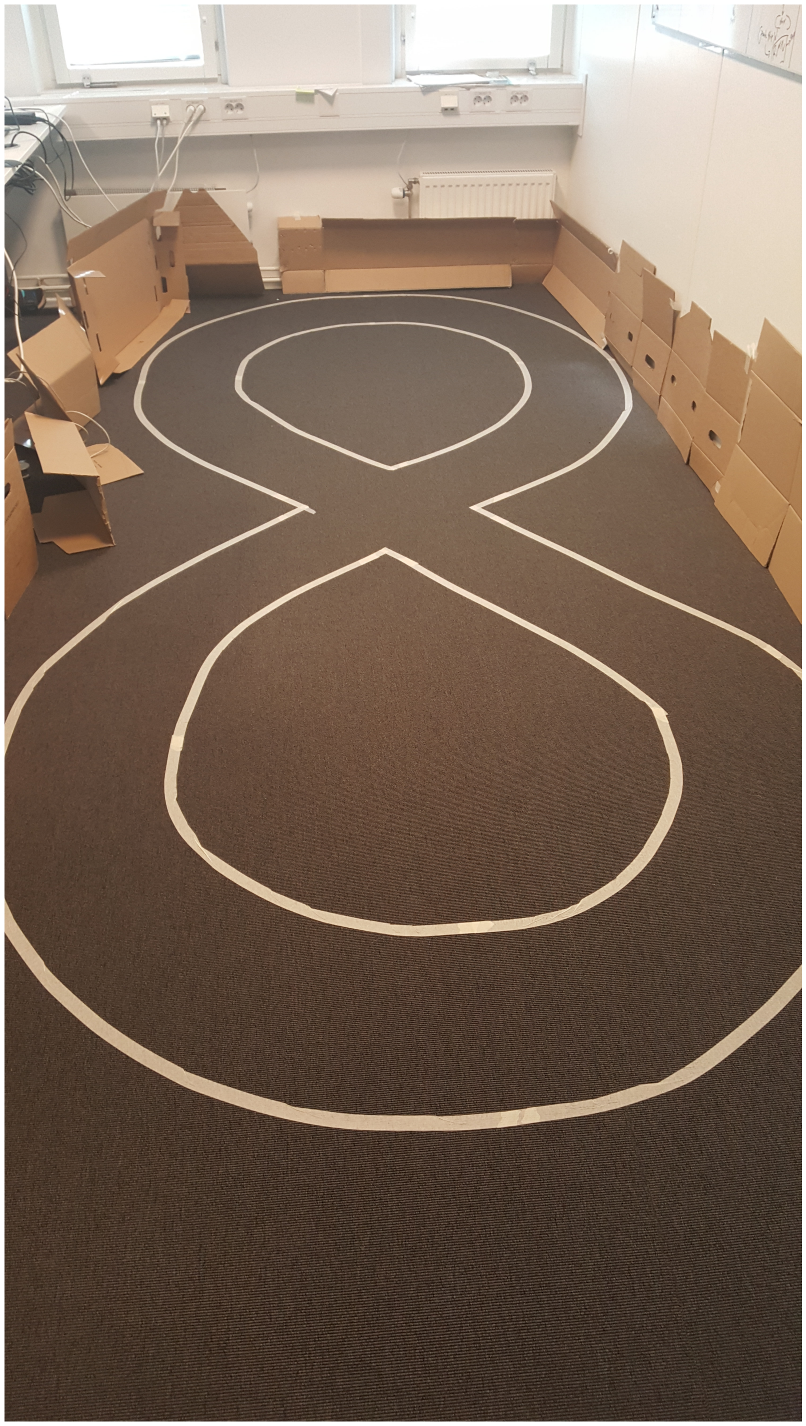


Figure 4.5: The eight track in which the general driving model was trained. The lane width of the track is 35cm.

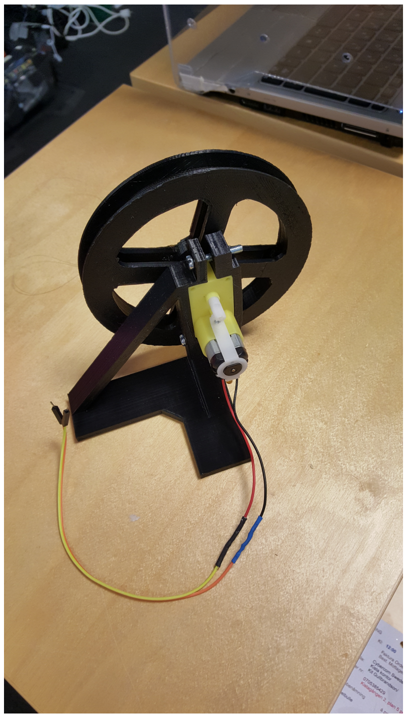


Figure 4.6: The spool to pull the leading object.

For the scenario 3 steering avoidance model a straight track was prepared with a width of 50 cm. A 10 cm pink cube was placed at differing range but always in the middle of the lane to be the object to avoid. The driver then drove towards the object to avoid it at different distances and at a 50/50 left/right rate. For scenario 3 39 thousand images were gathered doing this, which corresponds to 92 different data gathering runs, and were then combined with the general driving dataset.

Single-state Model

The single-state model was modelled as a CNN thanks to its strength when it comes to extract relevant features in images. There are no clear guidelines on how to design a neural network due to the variety of data used in each project. The main purpose of the single state model is to see where the lines are and to adjust so it is in the middle between the two. This could be seen as quite a few features that needs to be detected by the convolutional layer. Therefore attempts where done with simple network architectures consisting of a single convolutional layer followed by a few dense layers. This network was tested on both grey scale images and true colour images. The network trained on true colour images proved to be significantly better. The vehicle performance from this basic network was not sufficient. It managed to predict a pretty good steering value when standing still but it did not manage to drive inside the track when the vehicle was moving.

Designing the single-state model was through trial and error assisted with the learning curve method. This led to a working network consisting of three convolutional layers followed by a couple of dense layers. To fine tune the network a script generating 100 random network architectures varying in the number of dense layers and the size of them was created. All the networks had the starting three convolutional layers with the same settings. Limits on the number of dense layers was 1-10 and the size of the layers was limited from 64-2048. The training took around 30 minutes per network training on around 77 thousand images. The chosen network for running scenario 1 had a test loss value of 2.09

but network architectures with test loss around 1.5 was generated by the script and then tested. These network performed worse in real life than the chosen network. All the networks were trained with the same data and tested with the same data. What the networks with lower loss value had in common compared to the chosen network was the size of the dense layers. The chosen one had 3 dense layers each with a size of 128 nodes. The network with lowest test loss value had 2 dense layers with a size of 1831 and 1251 nodes. The final network architecture for scenario 1 had three convolutional layers, followed by three dense layers with a size of 128 nodes each followed by an output layer. It is possible to look at the output of each filter in a convolutional layer. That is where the input image has gone through the filter and creating a new image showing what that filter is trained to detect. In Figure 4.7 one can see how different filters look in a convolutional filter when trained with the sequence 1 dataset.

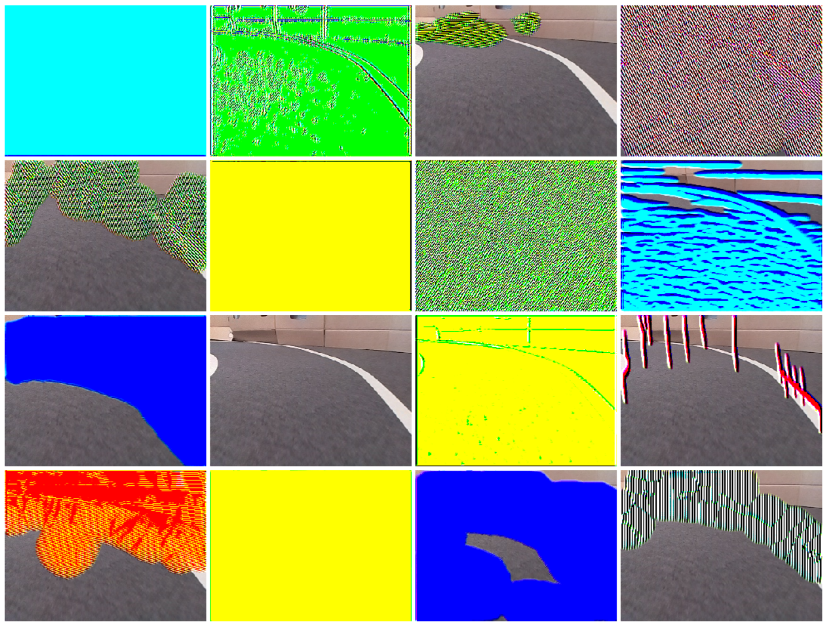


Figure 4.7: CNN filter output from a networked trained on the scenario 1 dataset

For scenario 2, keeping distance to an object, a new network had to be trained. For the network to output velocity instead of steering regression was chosen instead of classification. Choosing regression over classification should yield a seamless smoother speed control. This time the network was trained to draw conclusion based on how close and far away the pink box was. In Figure 4.8 two examples images can be seen on what the network should predict the vehicle speed on. Since the network architecture used for scenario 2 worked so well for analysing the image the three convolutional layers were used in this scenario as well. A few networks with different number of dense layer and sizes was tested and almost every configuration converged to a low loss value. The chosen network had 3 convolutional layer followed by four dense layers with 128 nodes each and an output layer.

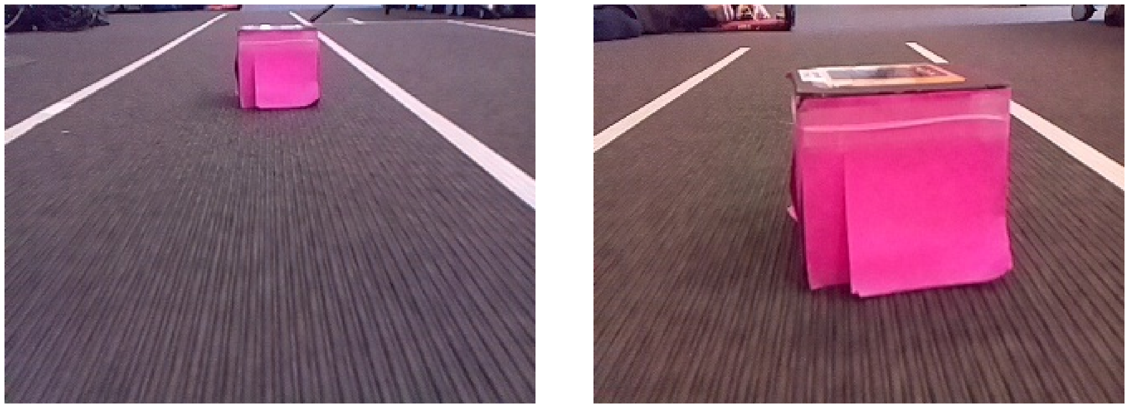


Figure 4.8: Two example images from Scenario 2 dataset

For scenario 3, going around an object, a new network had to be trained. The same architecture as in the scenario 1 was tested first since it worked so well for general driving. The test loss value for the network was around 3 and it did not manage to drive around the object at all. This was probably due to being able to detect and move around an object needs a network capable of extracting more features from the given image. The same script used earlier for generating random networks was used again for finding a network with low loss value. A network with a test loss value of 1.2 was generated and performed really well in real life. It had an architecture of 3 convolutional layers followed by two dense layers with a size of 911 and 778 nodes followed by an output layer. In Figure 4.9 one can see how different filters look in a convolutional filter when trained with the sequence 3 dataset. It clearly shows that it detects the box in multiple filters as well lanes.

A more detailed description of each model can be seen in Appendix A. Their learning curves can be seen in Appendix B.

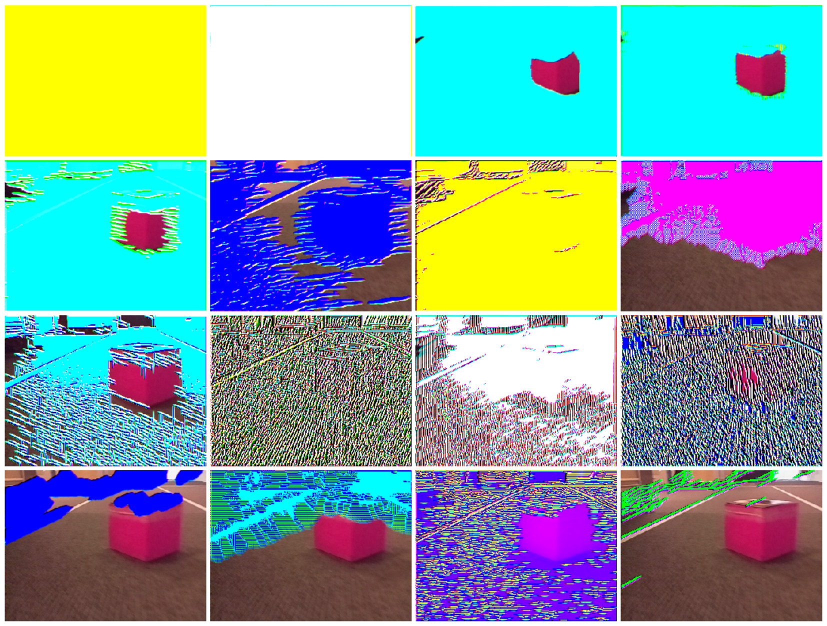


Figure 4.9: CNN filter output from a networked trained on the scenario 3 dataset

Sequential Model

The three different methods of doing a sequential model that was discussed in Methodology were attempted, firstly the naïve approach in which raw image sequence were used as input. This was at first confirmed functioning using the Mnist dataset [41] which is a set of hand written digits in the format of 28x28 binary images. This method was capable of learning a sequence of three pictures (1, 2, 3) and predict the next digit (4). An accuracy of 80% was quickly achieved.

However when presented a dataset with 160x120 truecolor images the networks were unable to converge and simply outputted the average steering of the dataset. This approach may still be possible but it was discared in this thesis. The next attempt was to pre-process the images using Canny Edges from the OpenCV2 library [42][43]. This simplified the image to a binary image in which only edges were visible, an example can be seen in Figure 4.11. This allowed the network to learn the problem quite well but still had a larger loss than the single-state model. A learning curve of this can be seen in Figure 4.12. This was also discarded as the results are no longer comparable with the single-state which improved from not using Canny Edges.

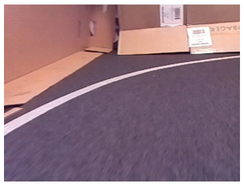


Figure 4.10: Example image.

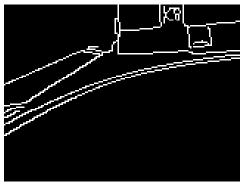


Figure 4.11: After Canny Edges.

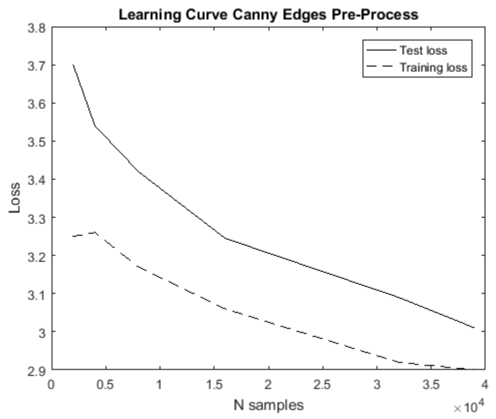


Figure 4.12: The learning curve when the sequence of images were pre-processed. In this specific image a sequence of 10 images was used.

Another method was to utilize a Keras wrapper which distributes and applies a neural network layer to a time-distributed sequence. So before the LSTM each image would be pre-processed by a convolutional, max pooling and flattening layer. This method proved to be very time-consuming with each epoch taking up to 20 minutes compared to a maxiumum of twenty seconds of the other methods. Additionally memory started to become an issue as it was no longer possible to hold all sequences of images. When unrolled into an array where each pixel is normalised to a 32-bit float, each 160x120 image held an approximate size of 2 MB. So with a sequence of 10 images in a dataset with 41 thousand images this becomes over 95 GB. It is possible to train the network on snippets of the dataset but this would make the process even more time consuming. However seen in Figure 4.13 the learning curve is decreasing very acutely and can possibly become the most successful. This method was discarded due to the additional complexity inherited in the method.

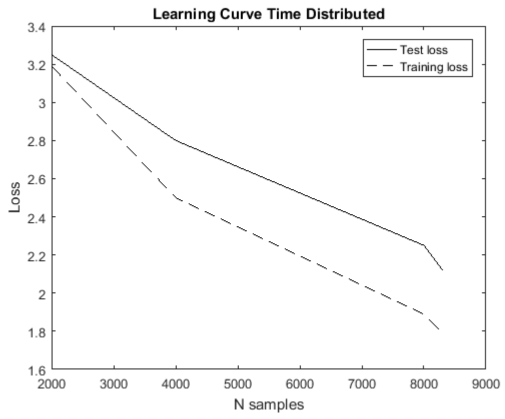


Figure 4.13: The learning curve with a time distributed CNN. In this specific image a sequence of 5 images was used. Not all datasets were used in this due to memory limitations.

The final and standing method was to use the single-state network which utilises the convolutional neural network. The layer before the output layer was redirected to become the output and then used to pre-process all sequences. The output then became a large vector of 'hidden values'. An LSTM was then trained on these sequences of 'hidden values' which proved to be fast, memory efficient and produced the lowest loss with possible exception of the time-distributed model.

Sequence Length

The question of how long a sequence should be was an open one. It was possible that a longer sequence holds more information or possibly that a too long a sequence is too many input parameters. As images were gathered in a frequency of 20 Hz the time between each image was 50 ms. To get a perspective images from a sequence of 50 can be seen in Figure 4.14. An overtaking from Scenario 3 took on average 80 images from start to finish.

To evaluate this problem an algorithm compiled 50 different networks all similar but on the input layer where they were modified to take sequences from 1 up to 50 images. The models were trained on their complementary data and evaluated. In Figure 4.15 a clear improvement can be seen on test loss for the LSTM and BLSTM and a neglibible improvement for the SimpleRNN. The same behaviour can be seen in Figure 4.17 however less clear due to the quick convergance. In 4.16 the test loss does not improve with an increased sequence length. This could either be due to it being regression or simply that the model converges extremely fast.

Another effect of an increased sequence length that needs to be considered is that it may reduce the amount of samples if it is pieced together manually as done here. Here a dataset consists of a serie of imagesets. Each imageset have been gathered separately at different times. For Scenario 1 they are rather few and long, averaging about 1250 images per set. But Scenario 2 and 3 averages at 200 and 230 per set respectively due to the nature in which they are gathered. If a sequence of length n consists of image 1, 2...,n and 2, 3..., n+1 until n reaches the size of the set. One set will then be able to produce a number of datapoints equal to $points = setsize - n$. So when doing a sequence of 50 from a set of 200 each set will only be able to produce 150 datapoints. This effect reduces the datapoints

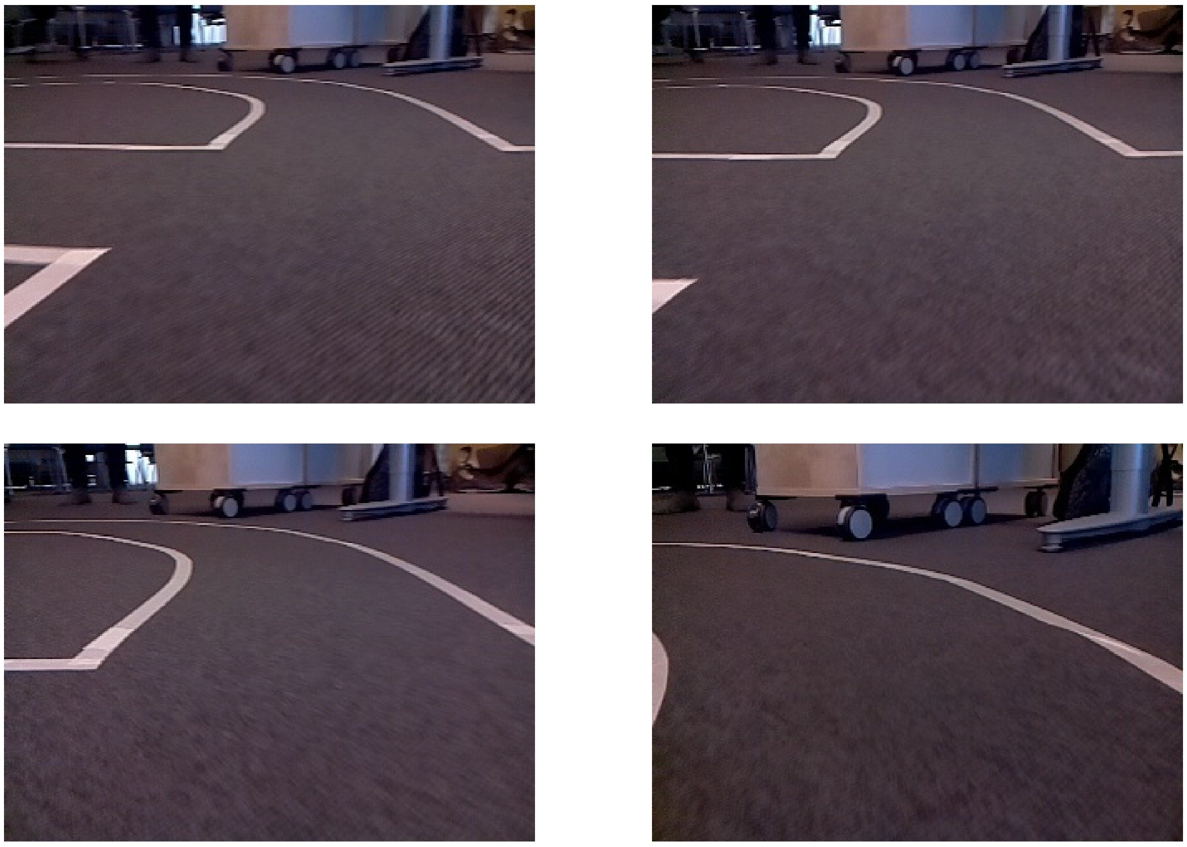


Figure 4.14: Four images from a sequence of 50. The images are the 1st, 5th, 15th and 50th respectively.

in scenario 3 from 34 000 images to 30 000 sequences. The effect of this can be seen in Figure 4.17 on the small upswing in test error at the end.

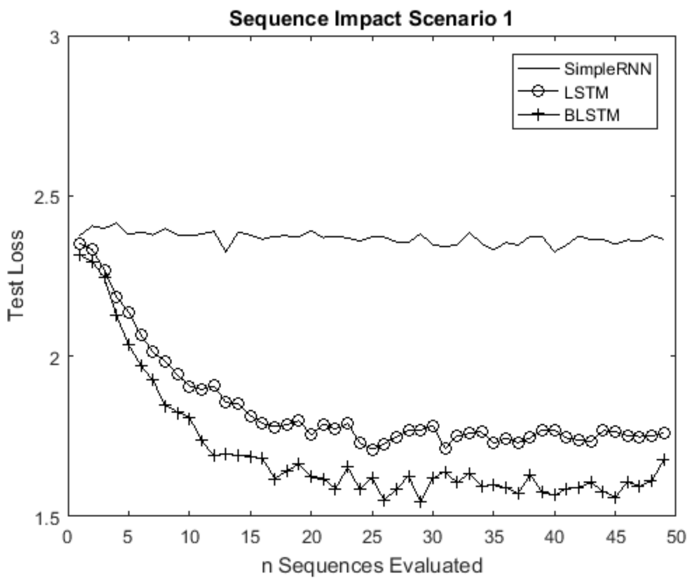


Figure 4.15: Training 50 networks with differing sequence length in Scenario 1, general driving.

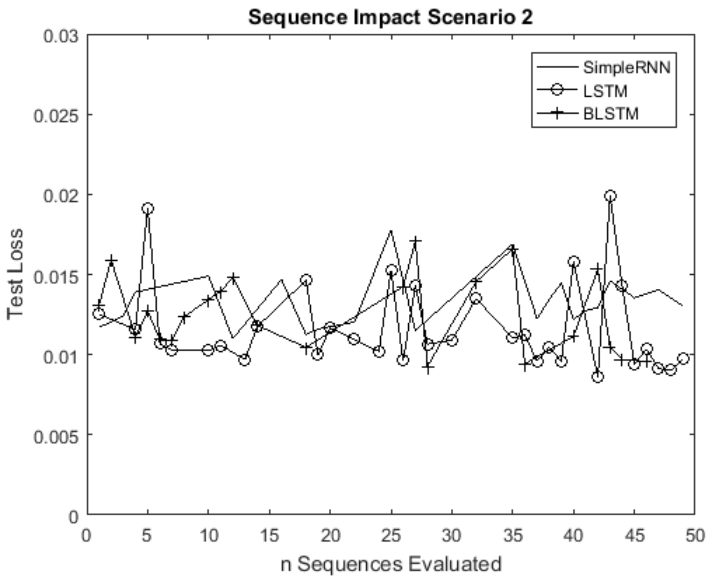


Figure 4.16: Training 50 networks with differing sequence length in Scenario 2, distance control.

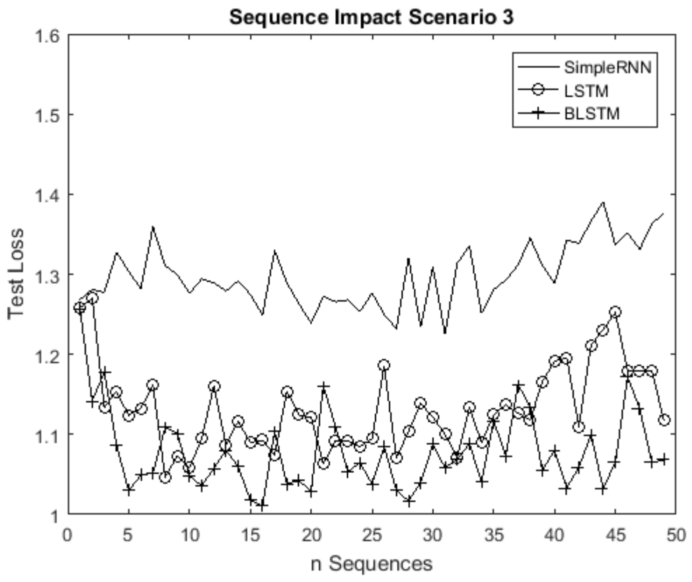


Figure 4.17: Training 50 networks with differing sequence length Scenario 3, an overtaking.

Final sequential models

Based on these tests and vigorous experimentation six models were designed. All used a single BLSTM layer as it reliably showed the best test result and the extra computational time did prove not to be a limiting factor. The input to the network is the output from the single-state model's penultimate layer for the same scenario. The size of that input differed from scenario to scenario. Scenario 1 and 2 had an input of 128, Scenario 3 had and input of 778. The amount of dense layers and their sizes varied between the models to improve test loss. All dense layers used ReLU as activation function with exception of the final layers in the categorical models where Softmax is used as recommended. The models can be seen in Appendix A. The learning curves of the models can be seen in Appendix B

4.5 Computation time

The computation time for one cycle is measured from the time of the image capture to the output from the neural network. The neural network predictions are run on the laptop which is used for driving the car using its dedicated GPU, a GeForce 840m. The time for the Raspberry Pi to send the steering and motor request to the Arduino over serial communication is disregarded due to not being able to measure it. The measurements are an average of 3000 samples, the time is in milliseconds. The maximum time measured in the samples are also presented. The results can be seen in Table 4.1.

Table 4.1: Average time and maximum time for a computation cycle for each network and scenario

Scenario 1	**Single State**	**Sequence 5**	**Sequence 25**
Average time [ms]	31.2	31.5	31.9
Max time [ms]	66.2	58.8	58.2

Scenario 2	**Single State**	**Sequence 5**	**Sequence 25**
Average time [ms]	31.3	30.8	31.9
Max time [ms]	59.1	56.7	55.9

Scenario 3	**Single State**	**Sequence 5**	**Sequence 25**
Average time [ms]	30.9	31.1	31.7
Max time [ms]	54.2	66.1	56.3

The computation time only for the neural network prediction can be seen in Table 4.2. The predictions are run on a GeForce 840m. The measurements are an average of 3000 samples, the time is in milliseconds.

Table 4.2: Average time for a neural network prediction for each network and scenario

Average time [ms]	**Single State**	**Sequence 5**	**Sequence 25**
Scenario 1	2.1	5.7	12.9
Scenario 2	2.4	6.0	13.4
Scenario 3	4.4	9.3	16.2

This clearly shows that the bottleneck in a computation cycle is not the neural network predictions but rather something else. What takes so long time compared to the prediction is probably the transfer of the image from the Raspberry Pi 3 to the laptop which is done over WiFi using the built-in WiFi module on the Raspberry Pi 3.

Chapter 5

Result

The three different scenarios where tested on three different models and a human driving the vehicle. The models were a single state model, multiple-state with a sequence of 5 images and a multiple-state model with a sequence of 25 images.

To assist in the analyse of the test results an average curve was generated from each test run. This was done by taking the average value for each time step.

5.1 Scenario 1

Scenario 1 was run on an 4x1.8m eight-track with a track width of 35 centimetres. The vehicle had a speed of 0.4 m/s. Each run began on the same spot with it's location and angle fixed to increase the repeatability of gyro data. Four different models were tested; single state model, multiple-state with sequence 5, multiple-state with sequence 25 and ideal driving. The ideal driving is when a human drives as good as possible to stay in the middle of the lane. Each setup was tested until 30 successful runs were made. The result can be seen in Table 5.1. In terms of success rate the single state model performs better than the sequential ones.

Table 5.1: Result of the networks for Scenario 1

	Fails
Single	0
Sequence 5	4
Sequence 25	2

Each models driving path and its average can be seen in Figure 5.1, only done on the 30 successful runs. The path data acquired using the velocity of the car and gyroscope data from the IMU. The starting point is marked with a vertical black line.

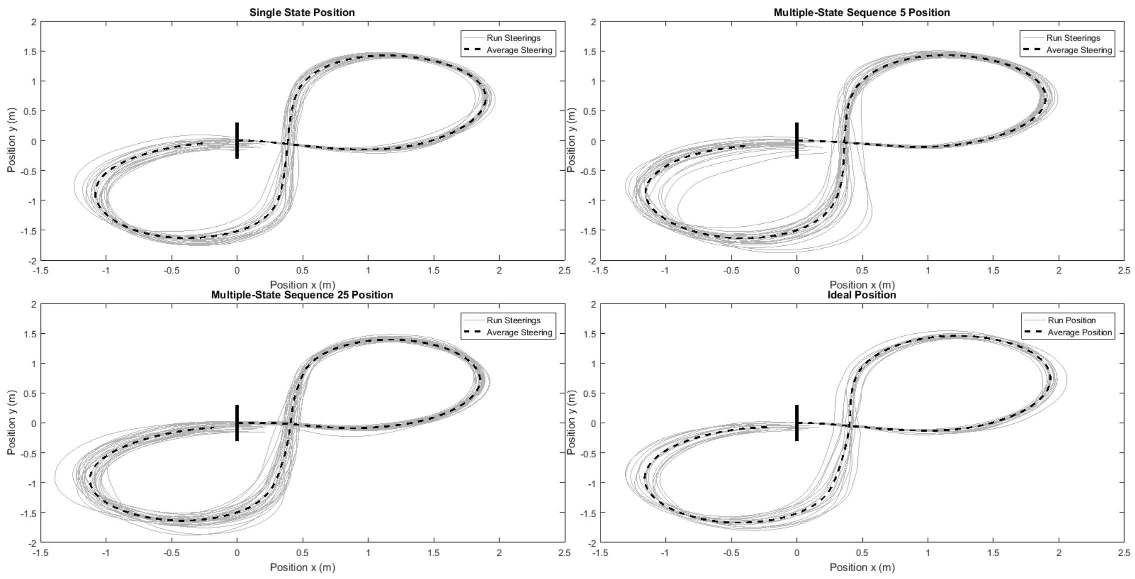


Figure 5.1: Driving path of each model

The standard deviation in X and Y position for each model can be seen in Figure 5.2, only done on the 30 sucessful runs. The standard deviation is calculated by using each models average position curve. The result shows that the single-state model run most similar each time which also can be seen in Figure 5.1 when comparing the plots between each other.

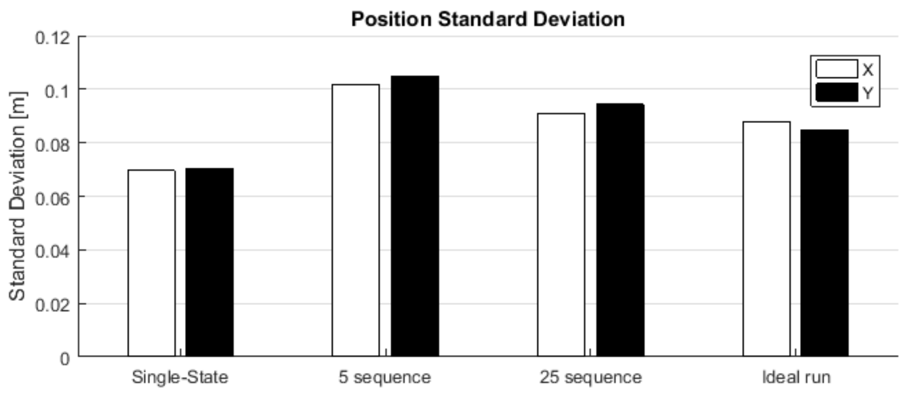


Figure 5.2: Standard deviation of the position for each model

The neural networks output a steering request to the servo. Each networks output, and the ideal from the joystick, together with each models average curve for each model can be seen in Figure 5.3, only done on the 30 succesful runs. When a steering requests deviates alot from the average it means that the network is making a wrongful prediction. This behaviour is slightly more visible in the models with a sequence 5 and 25 which also can be seen when looking at the standard deviation of the steering requests which is displayed in Figure 5.4.

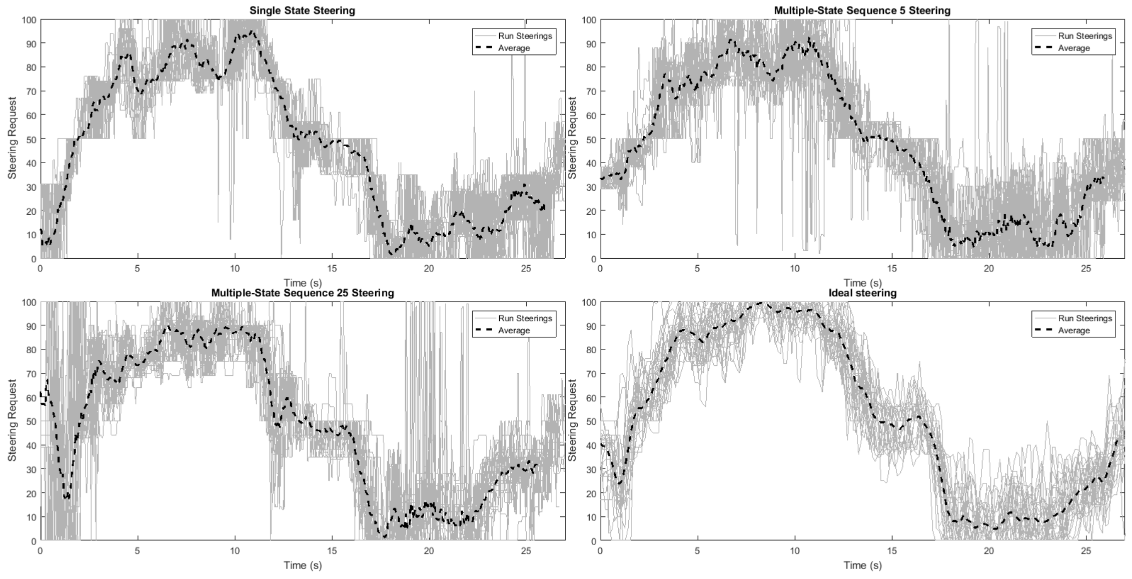


Figure 5.3: Steering request of each model

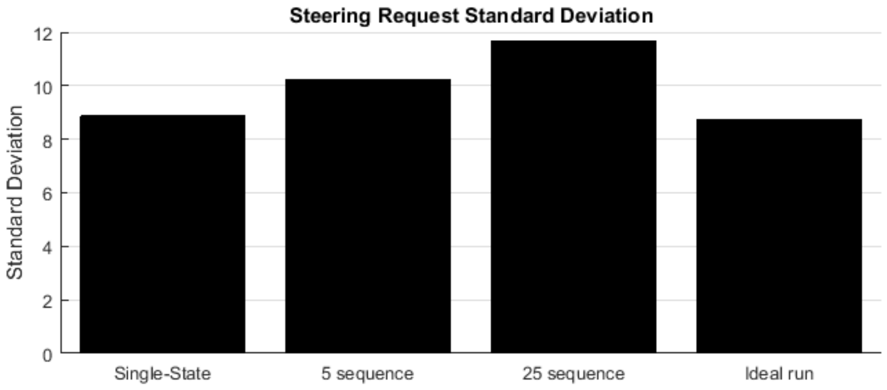


Figure 5.4: Standard deviation of the steering request for each model

Both the sequential models had failed runs. The output of a neural network in a failed test run can be seen in Figure 5.5. The figure also shows the position of the car and at what time it happens. The relevant information from the steering request plot is marked inside the dashed rectangle. There are many, large mispredictions around $time = 20$. The network outputs values up to 100 which corresponds to maximum turn to the left. That the vehicle turns to the left can be seen in the position plot.

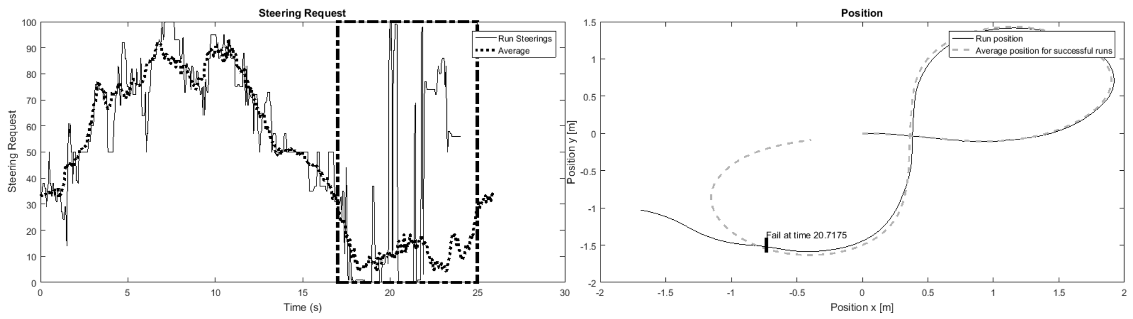


Figure 5.5: Steering request and vehicle position in a failed test run

5.2 Scenario 2

Scenario 2 was run on a 6.5 meter long track with an inner width of 42 cm centimetres. The vehicle was placed in the middle of the lane in the beginning of the track. An 9x9x9 centimetre pink box was placed 35 centimetres in front of the vehicle and in the middle of the lane. The box was connected with a fishing wire to a spool at the end of the track to pull the object towards it. During each test run the object was pulled with a velocity following a triangle wave. The distance to the object was measured with an ultrasonic sensor mounted on the front of the vehicle. The steering of the vehicle was done manually using a joystick to make sure the vehicle drove as straight as possible. The ideal run held its distance by measuring the distance with an ultrasonic sensor and regulating the vehicle velocity with a PID-controller. The ideal runs' reference distance was 50cm. Each model was run until 30 successful runs had been completed, but in this specific scenario there were no failures. In Figure 5.6 the distance to the object can be seen. In Figure 5.9 the motor request, which is the direct output of the neural network can be seen.

For the distance a straighter line indicates a constant distance and a good distance control. In contrast to the motor request where a sinusoidal pattern implies that the network is working to counteract the distance change. To numerate these the standard deviation of the curves have been calculated and can be seen in Figure 5.7 for the distance and Figure 5.10 for the motor request. Further to evaluate the straightness of both the distance and the motor request a least-square line was fit to each average and then the mean distance to the measured average was calulated and presented in Figure 5.8 for the distance and in Figure 5.11 for the motor request.

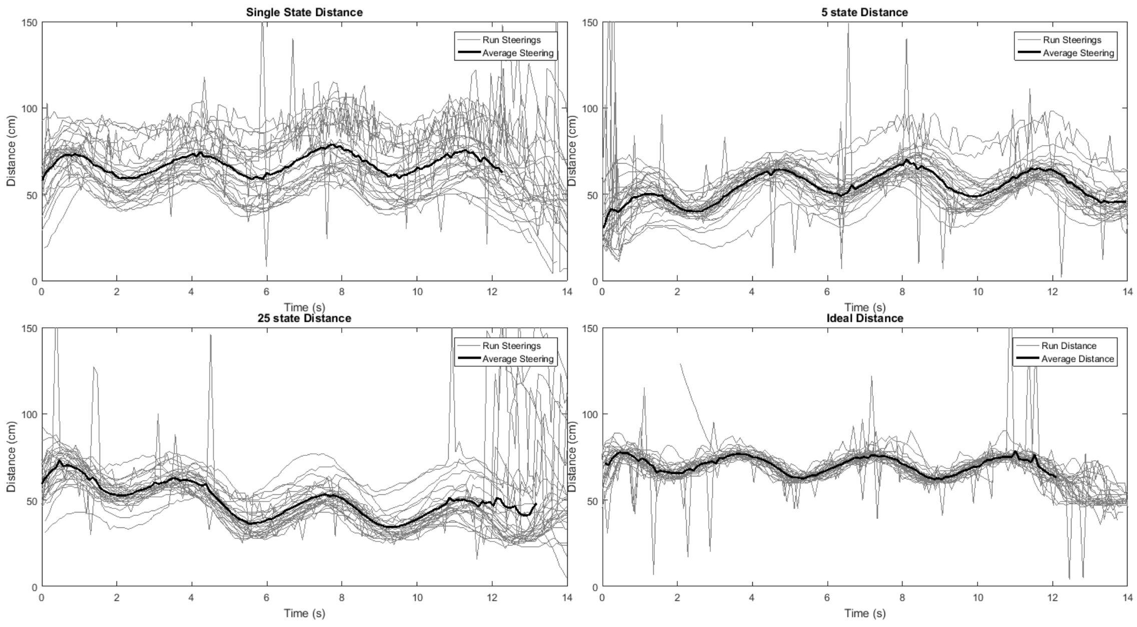


Figure 5.6: Measured distance to the cube ahead for all runs on the three different networks and from the ideal human driver.

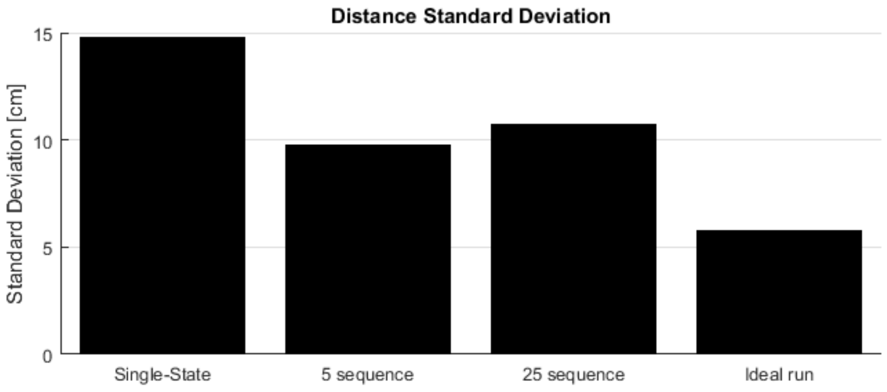


Figure 5.7: Standard deviation of the distance for each model and the ideal runs.

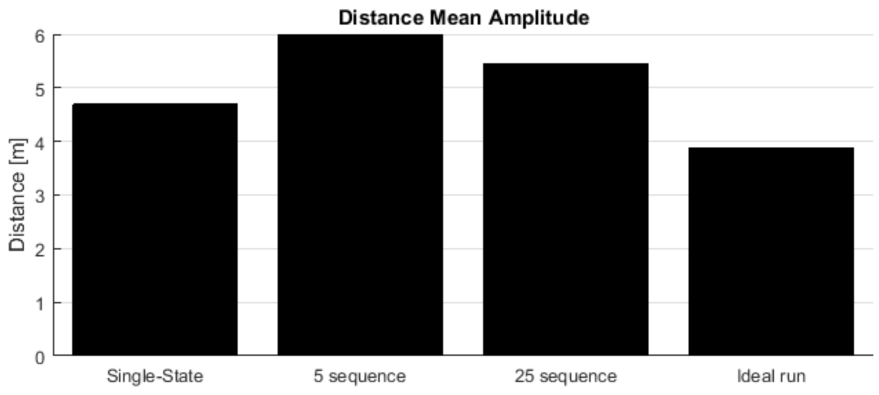


Figure 5.8: Amplitude of the distance for each model and the ideal runs.

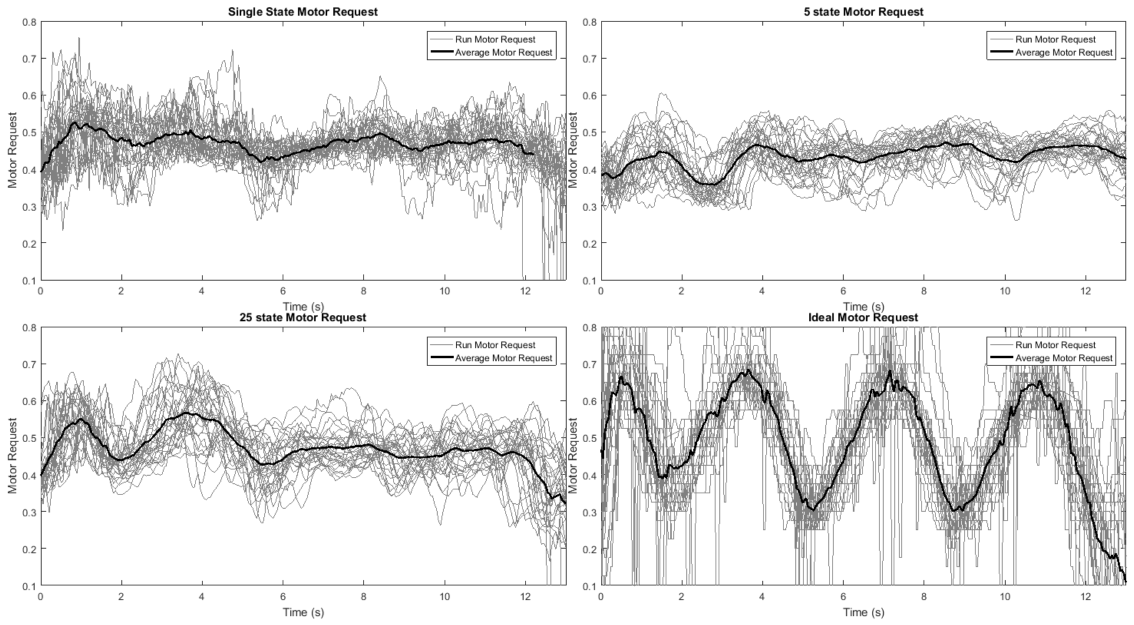


Figure 5.9: Motor requests for all runs on the three different networks and from the ideal human driver. A large number indicates a faster velocity.

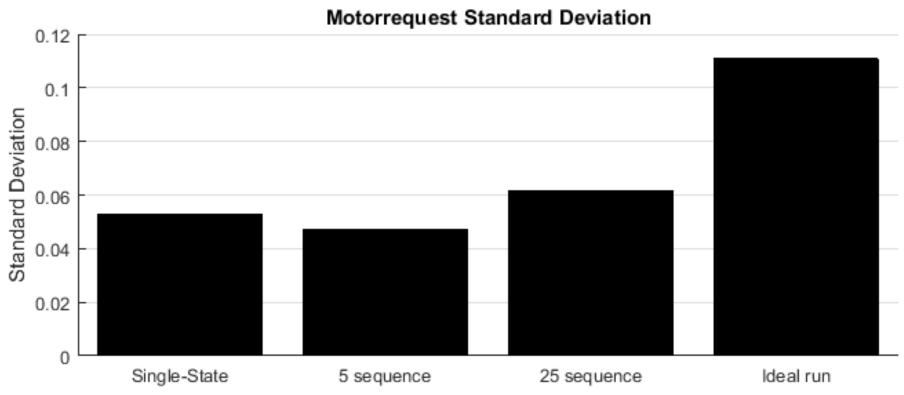


Figure 5.10: Standard deviation of the motor request for each model and the ideal runs.

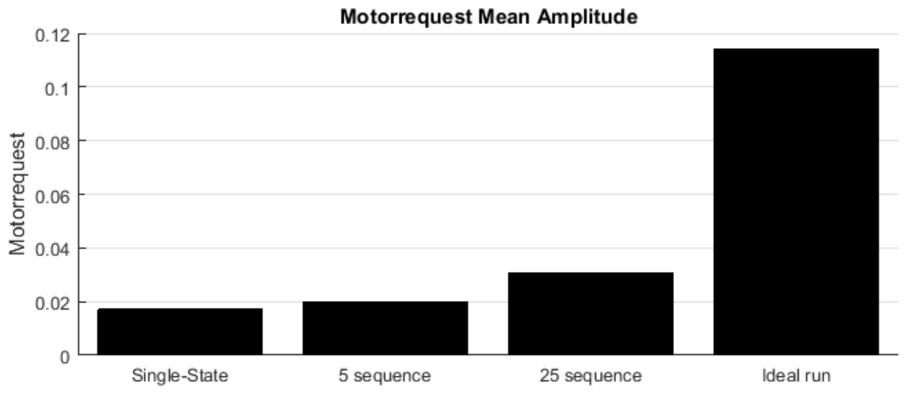


Figure 5.11: Amplitude of the motor request for each model and the ideal runs.

5.3 Scenario 3

Scenario 3 was run on a 6.5 meter long track with an inner width of 50 centimetres. A 9x9x9 centimetre pink box was placed in the middle of the lane, 3 metres from the starting point. The vehicle was placed in the middle of the road. The vehicle had an average speed of 0.4 m/s. Each model was tested 29 times. The test result can be seen in Table 5.2. The positions of all runs can be seen in Figure 5.12 and the standard deviation of the position in Figure 5.13. The standard deviation was only calculated on the right turns since they were the most plentiful.

The steering requests for right turns can be seen in Figure 5.14 and for left turns in Figure 5.16. The standard deviation of the steering request can be seen in Figure 5.15. The standard deviations was only calculated on right turns as they were the most plentiful.

Table 5.2: The number of failure and how many times the car chose to go around the object on the left side. The runs that did not go around the left side went on the right side.

Neural Network	**Fails**	**Left path**
Single	4	9/30
Sequence 5	1	7/30
Sequence 25	3	2/30

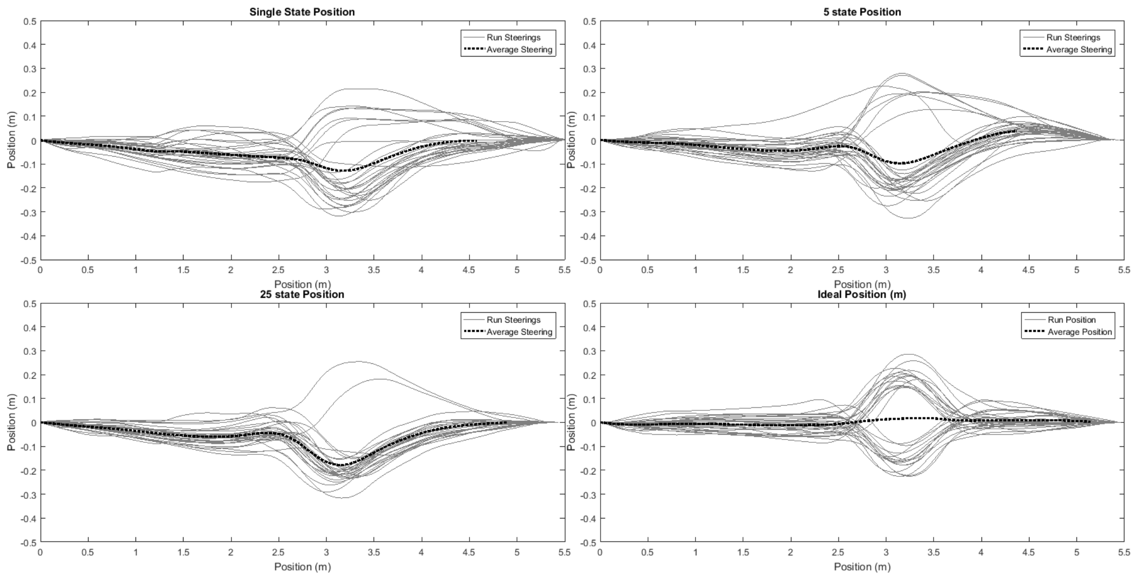


Figure 5.12: Measured position for all runs on the three different networks and from the ideal human driver. Leftmost of the plot is the starting position.

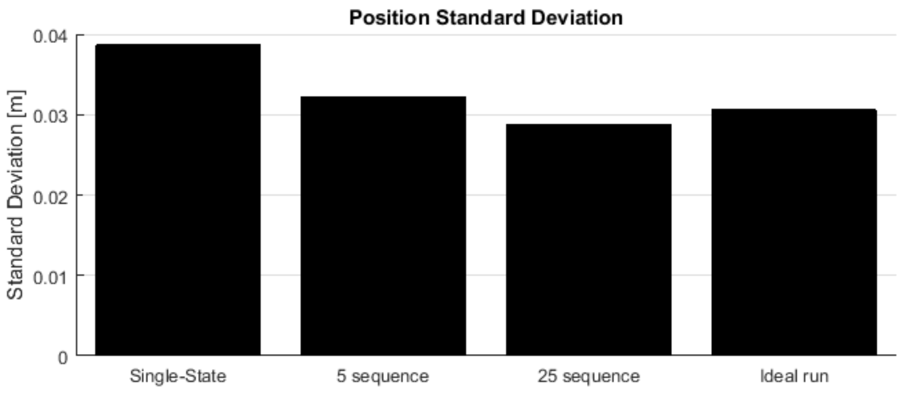


Figure 5.13: Standard deviation of the position for each model and the ideal runs. This is only calculated on right turns.

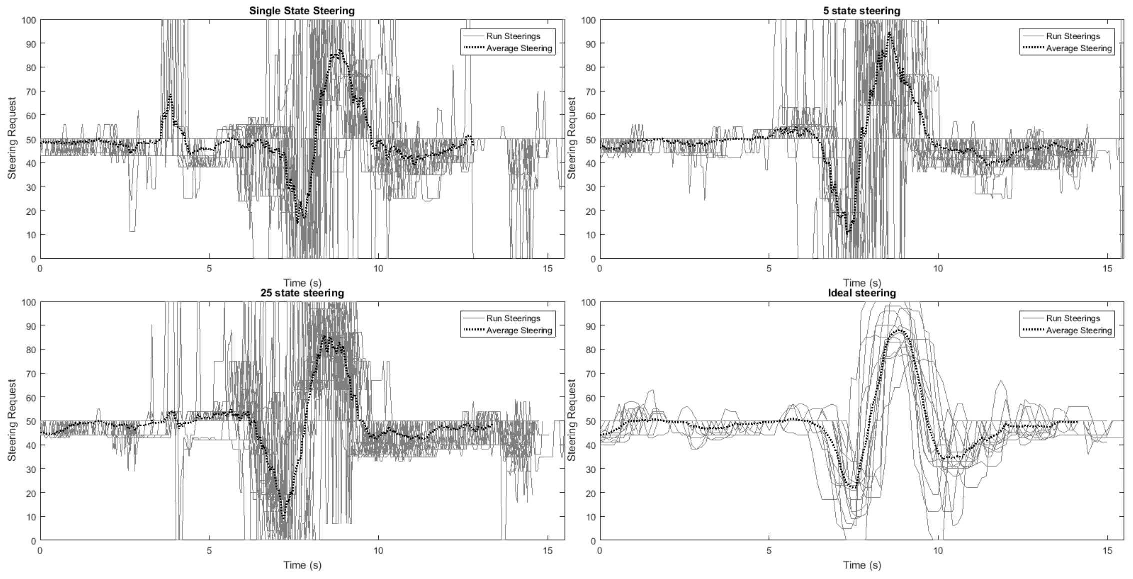


Figure 5.14: Steering request for all right turn runs on the three different networks and from the ideal human driver.

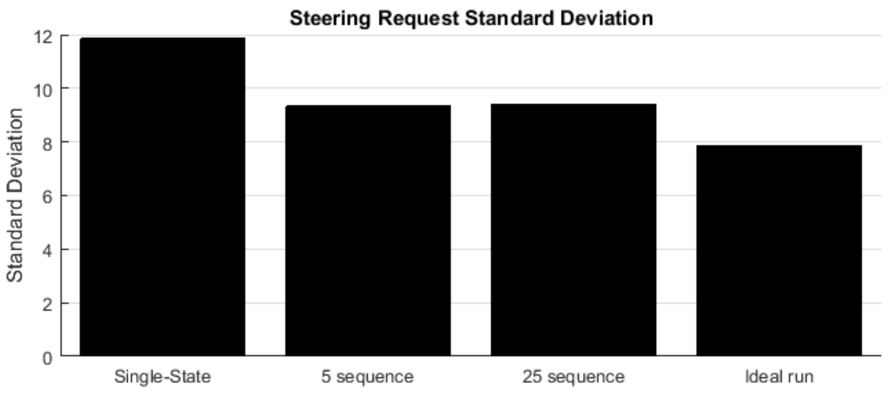


Figure 5.15: Standard deviation of the steering request for all right turn runs on the three different networks and from the ideal human driver.

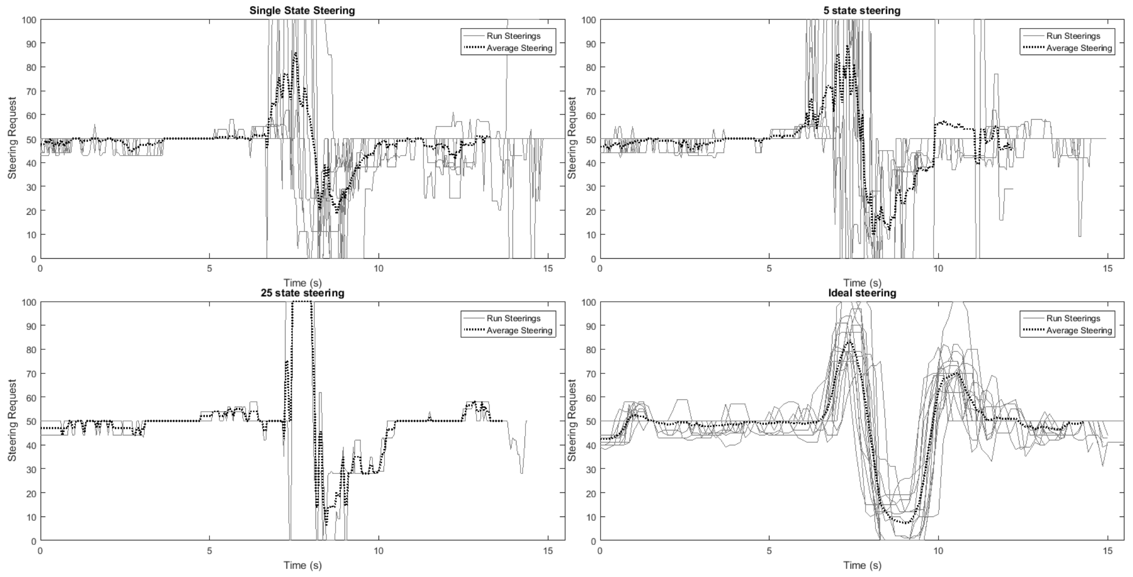


Figure 5.16: Steering request for all left turn runs on the three different networks and from the ideal human driver.

All networks had failures in this scenario. They were defined as a failure if they strayed off the road and was unable to recover, or if they collided with the object. Three example failures of colliding into an object can be seen for single-state in Figure 5.17, for the 5-state model in Figure 5.18 and finally for the 25-state model in Figure 5.19. All these share an undecisiveness during the brief moment where the car has to turn, around 6-8 seconds in.

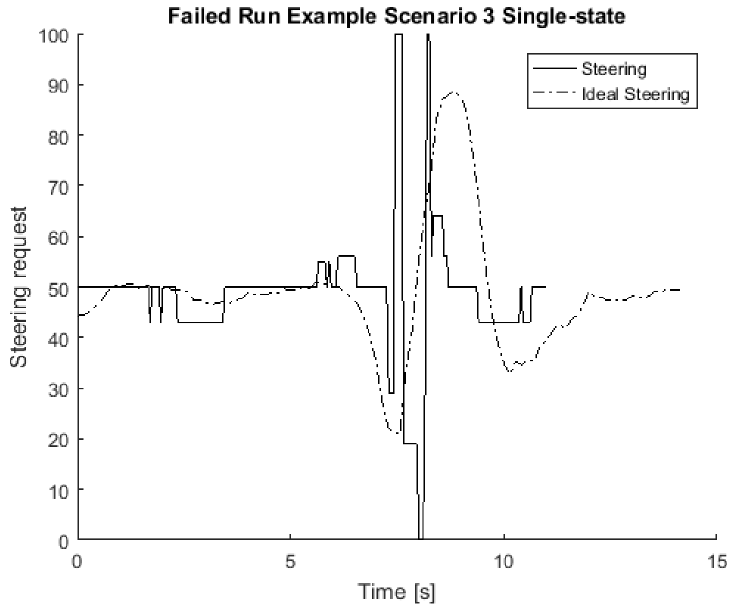


Figure 5.17: Steering of one of the failed runs in scenario 3 with a single-state model. Compared with the mean ideal steering of a right turn.

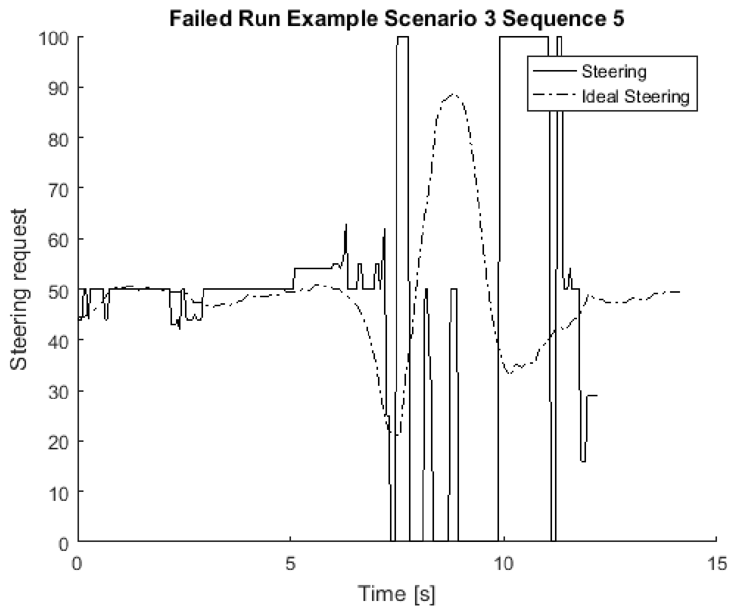


Figure 5.18: Steering of one of the failed runs in scenario 3 with a sequence 5 model. Compared with the mean ideal steering of a right turn.

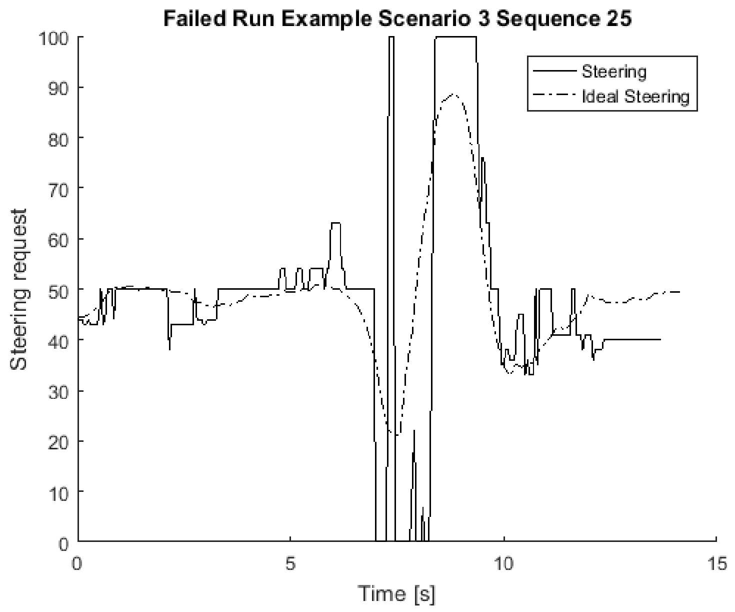


Figure 5.19: Steering of one of the failed runs in scenario 3 with a sequence 25 model. Compared with the mean ideal steering of a right turn.

Chapter 6

Discussion & Conclusion

6.1 Discussions

Looking at the results, how does the multiple-state method perform compare to the traditional single-state model? Had one only looked at the theoretical test loss this would have been clear, the multiple-state model wins hands down. But when applied in reality the results suddenly become less clear, as it is not always good, and when it is, not to the degree which the test loss shows, but why?

First, let's discuss some error sources in the tests themselves. All scenarios had the vehicle placed by hand on a marked spot on the track. The vehicle was placed at least 120 times for each scenario and if the starting angle of the vehicle were to deviate slightly from the decided placement it would lead to shifted position plot for the result. This is due to the lack of a global positioning system. We assume the starting position is exactly x=0, y=0 and the vehicle angle $\theta = 0$ but in reality this could differ slightly. Using a global positioning system could fix this potential error source. Another issue was the fact that the testing environment was not perfectly repeatable, small differences in the environment could create small differences in the result. This was counteracted as best as possible by covering all open space.

Another loose variable between the scenarios is the amount of training data used for each. What if the multiple-state model works better when there is less data available, and therefore the results are the way they are? Additionally ponder that a dataset composed all of copies of the same datapoint would not yield an equally good model as a dataset with unique datapoints. A good blend of datapoints is required to cover all possible cases and to create a model that generalises well. We have not been able to figure out if the datapoints from one scenario held more information that the datapoints from another. It may have due to the nature of the scenario, it may also have in the way the datapoints were gathered which was different between each scenario.

When analysing the scenarios the standard deviation of the output is analysed. The reason is that given practically the same scenario each network should give practically the same outcome. If it does not it's a measure of variance issues. If it does, and the result is poor, it's bias issues. Each scenario had their own peculiarities, let's look at them one at a time.

Scenario 1

The results for scenario 1 shows that the multiple-state models does not outperform the single-state model. The lack of consistency in the runs can both be seen visually in Figure 5.1 where the multiple-state networks has many more different ways of driving around the path. It can also be seen in the calculated standard deviation for both the position and steering requests. Looking at the steerings for the different model the multiple-state models both have more outliers which often are faulty decisions. This continues to show in the raw number of failed runs where the multiple-state models failed to complete the run and drove off the track.

Looking at the steering and the example of a failed run most of the outliers happened in the same two locations, one at 10 seconds in and another at 20 seconds in. What happened here in reality can be seen in Figure 4.5. On half the track, right, the background was a constant wall, on the other half, left, the background was slightly fluctuating open space. The failures started happening when the car was mostly turned to the open space. This implies that the multiple-state networks failed to generalise as well as the single-state model and had issues in dealing with a non-constant background. This in turn implies overfitting on the multiple-state networks part.

One conflicting issue here is that the theoretical test loss for the multiple-state network, which can be seen in Appendix B, is much lower meaning it could guess more correctly on the test pictures. Why did it not outperform the single-state network? An answer to this could be overfitting. The output from the single-state network is only 128 compared to scenario 3's 778. It may just be that the model overfitted on such a small number of parameters.

The most probable reason to the bad performance is insufficient training data. The multiple-state networks fail on the part which faces open-office environment. This happens as it has not learned to discard irrelevant data properly. This may be as the data itself never changes in any particular manner when gathered. All data was gathered in two hour-long sessions which will result in a lot of the data being very similar and not covering all possible cases of what it can look like. A better approach would have to spread it out over multiple sessions to cover more of the ever-changing environment. This again implies variance issues with the multiple-state networks.

Scenario 2

It is is possible to visually see the increased consistency in the multiple-state models distance measurements. The standard deviation calculations confirms the difference. However looking at the motor request which is the direct output of the network the difference is not as clear. In combination with the fact that none of the networks have kept distance to the object ahead to a satisfactory degree, the results are hard to read. It is not possible to evaluate any behavioural differences, that the multiple-state model could see the direction of the movement.

Looking at the amplitude of the motor requests can suggest that the 25-state model performs more and has learned the problem better. The standard deviation of the motor request is unhelpful. The ideal run has a massive standard deviation in the motor request which can be credited to it not being a human driver. Also a low standard deviation here does more imply a consistently faulty answer.

In summary; the scenario hints that the multiple-state network has learned the problem better, but the scenario by construct is not viable enough to establish any result. The networks need more training data and the ideal run needs to be made by the same way in which the scenario's dataset was gathered, in this case run by a human driver.

Scenario 3

The initial theory in the overtaking scenario was that the single-state model would have issues deciding on which path to take, and that the multiple-state models would overcome this. This was not the case as the single-state model did only failed 4 runs. But it did have a more ambivalent approach to which path around the object to take as can be seen in Figure 5.12. Combined with the improved standard deviation from the multiple-state models it implies that the multiple-state models learned the problem better and can generalise better. The test loss for the multiple-state model is also clearly less which can be seen in Appendix B which again confirms the result of this scenario.

The most interesting point in this scenario is that there does not seem to be any behavioural differences. As can be seen in Figure 5.18 and Figure 5.19 the multiple-state network fails by conflicting decisions just like the single-state network does.

Hardware Comparison

Multiple-state networks can be more or less memory intensive depending on the method used. The one used in this thesis is significantly more memory efficient as it operates on the output of a convolutional neural network of over a hundred features whilst a single 120x160 image holds over fifty thousand features. This also shortens down training time from up to a couple of hours to between a few seconds to a minute. If other methods are used such as time-distributed convolutional neural network then the memory usage and time consumption will become unmanageable by normal means and will require special hardware or software. The single-state network does not use as much memory as it manages single units and not a sequence. Additionally since the multiple-state model used in this thesis requires its data to be processed by a single-state network to be trained well first, the training time will always be longer than the single-state.

A single state network is predicts much faster compared to a multiple-state network with a sequence of 5 and 25 images which can be seen in Table 4.1 and 4.2. A multiple-state network with a sequence of 5 takes over twice as long time and with a sequence of 25 it could take up to over 6 times as long time. This shows that the longer the sequence the more powerful computation power is needed for keeping the same prediction speed as for a single-state network. It needs to be noted that the multiple-state model's timing includes additional steps beyond just two predictions, that of function calls and variable assignment which may differ between systems and implementation methods. However it is deemed to small that it is irrelevant for the argument.

A computation cycle took around 31.2 milliseconds, which would be a rate of around 30 frames per second, which the largest part of that time is the transfer of the image from the Raspberry Pi to the laptop over Wi-Fi. To reduce that time one could do the prediction directly on the embedded device. That would however put higher requirement of the processing power of the embedded device that could compete with a GeForce 840m GPU. We tried for instance running the vehicle using another laptop with an Intel i7 processor but we only managed to get a rate of 5 computation cycles per second which made the vehicles performance unusable. From this we drew the conclusion that we would get an even worse performance if we were to run the predictions on the Raspberry Pi 3 since its processing power is much lower compared to the laptop and was therefore not tested. Another way to reduce the transfer speed would be to use an external Wi-Fi adaptor for the Raspberry Pi using the much faster 802.11ac protocol, instead of the built in Wi-Fi module using the 802.11n protocol, and connect it to the router via the 5 GHz band. This would theoretically increase the transfer speed and therefore make it possible to run the car with a faster update rate.

6.2 Conclusion

With little doubt, the multiple-state model a viable on-par alternative to the single-state model. On paper it learns all scenarios faster and better than the single-state model. However it is not necessarily always better in reality and it is not yet made clear what decides if it is superior or not. Possibilities are that the multiple-state model learns faster and therefore has tendencies to overfitting if not cared for. Another possibility is that it is less effective in scenarios with less sequential information. There does not seem to be any inherent behavioural differences, or at least, none that can be separated from simple performance differences.

Based on these test results there are no behavioural differences, but this may just be as they not visible enough in the scenarios presented. To evaluate the behavioural differences properly more edge-case scenarios and more repeatable environment is required, for example in a simulation.

There are differences in hardware requirements depending on how the multiple-state model gets its inputs. The model finally used in this thesis that is based on the single-state model will always take more time to train and to predict with as it inherently uses the single-state model as a part of it's algorithm. In reality the single-state model is at least twice as fast. The difference in speed is overshadowed by other delays such as communication and may often be neglected. Additionally if any other method is used there will be extra considerations required in terms of memory consumption as the multiple-state may take up to the number of sequences times the memory for each datapoint.

In short, if a single-state model is already trained and the data may hold sequential information, making it into a multiple-state model is simple and often worth the try.

6.3 Future Work

Recurrent Neural Networks: More experimentation is required before it's possible to clearly say if, when and why the multiple-state networks are a step forward. Future experiments need to be have a more controlled testing environment, such as a simulation. Additionally more scenarios needs to be analysed as it is still unclear when the multiple-state networks are better.

Analysing effect of hardware variables: Utilising more variables than the image, such as velocity, steering, gyro, accelerometer and ultrasonic sensors may prove to be effective. When driving a person uses much more than just their eyes to make a decision.

Trajectory planning with Recurrent Neural Networks: There is a possibility to not only teach the model a single action based on a state, but a whole sequence of actions. It would be interesting to see if the model could do correct long-term predictions or if these sequences could be combined to a set of average actions offering additional stability.

Analysing time-distributed neural network: In this thesis the time-distributed convolutional neural network method was discarded due to its complexity. It did imply a very good result and should be analysed more to see if this method will yield different results that the one presented in this thesis.

Bibliography

[1] National Highway Traffic Safety Administration et al. National motor vehicle crash causation survey: Report to congress. *National Highway Traffic Safety Administration Technical Report DOT HS*, 811:059, 2008.

[2] Bel Geddes. *Magic Motorways.* Random House, New York, 1940.

[3] Dean A Pomerleau. Knowledge-based training of artificial neural networks for autonomous robot driving. In *Robot learning*, pages 19–43. Springer, 1993.

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

[5] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[6] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, et al. Junior: The stanford entry in the urban challenge. *Journal of field Robotics*, 25(9):569–597, 2008.

[7] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, MN Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[8] Charles Reinholtz, Thomas Alberi, David Anderson, Andrew Bacha, Cheryl Bauman, Stephen Cacciola, Patrick Currier, Aaron Dalton, Jesse Farmer, Ruel Faruque, et al. Darpa urban challenge technical paper. *Journal of Aerospace Computing, Information, and Communication*, 2007.

[9] Hyunggi Cho, Young-Woo Seo, BVK Vijaya Kumar, and Ragunathan Raj Rajkumar. A multi-sensor fusion system for moving object detection and tracking in urban driving environments. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 1836–1843. IEEE, 2014.

[10] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, et al. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*, 2015.

[11] Jan Koutník, Giuseppe Cuccu, Jürgen Schmidhuber, and Faustino Gomez. Evolving large-scale neural networks for vision-based torcs. *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1061–1068, 2013.

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

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

[14] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[15] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[16] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[17] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

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

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

[20] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Aistats*, volume 15, page 275, 2011.

[21] K. Fukushima, S. Miyake, and T. Ito. Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):826–834, Sept 1983.

[22] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[23] R. Hecht-Nielsen. Theory of the backpropagation neural network. In *International 1989 Joint Conference on Neural Networks*, pages 593–605 vol.1, 1989.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[25] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *International Conference on Artificial Neural Networks*, pages 92–101. Springer, 2010.

[26] Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence*, 31(5):855–868, 2009.

[27] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[28] Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5528–5531. IEEE, 2011.

[29] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[30] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[31] Kazuya Kawakami. *Supervised Sequence Labelling with Recurrent Neural Networks.* Springer-Verlag, 2012.

[32] N. Yazdi, F. Ayazi, and K. Najafi. Micromachined inertial sensors. *Proceedings of the IEEE*, 86(8):1640–1659, Aug 1998.

[33] A Zul Azfar and Desa Hazry. A simple approach on implementing imu sensor fusion in pid controller for stabilizing quadrotor flight control. In *Signal Processing and its Applications (CSPA), 2011 IEEE 7th International Colloquium on*, pages 28–32. IEEE, 2011.

[34] Paul D Groves. *Principles of GNSS, inertial, and multisensor integrated navigation systems.* Artech house, 2013.

[35] AD King. Inertial navigation-forty years of evolution. *GEC review*, 13(3):140–149, 1998.

[36] François Chollet. Keras. `https://github.com/fchollet/keras`, 2015.

[37] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[38] rosserial arduino - ROS Wiki. `http://wiki.ros.org/rosserial_arduino`. Accessed: 2017-05-18.

[39] MPU-9250 Breakout Github. `https://github.com/sparkfun/MPU-9250_Breakout`. Accessed: 2017-04-12.

[40] Raspberry Pi 3 Datasheet. `https://www.raspberrypi.org/documentation/hardware/computemodule/RPI-CM-DATASHEET-V1_0.pdf`. Accessed: 2017-05-18.

[41] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. -, 2010.

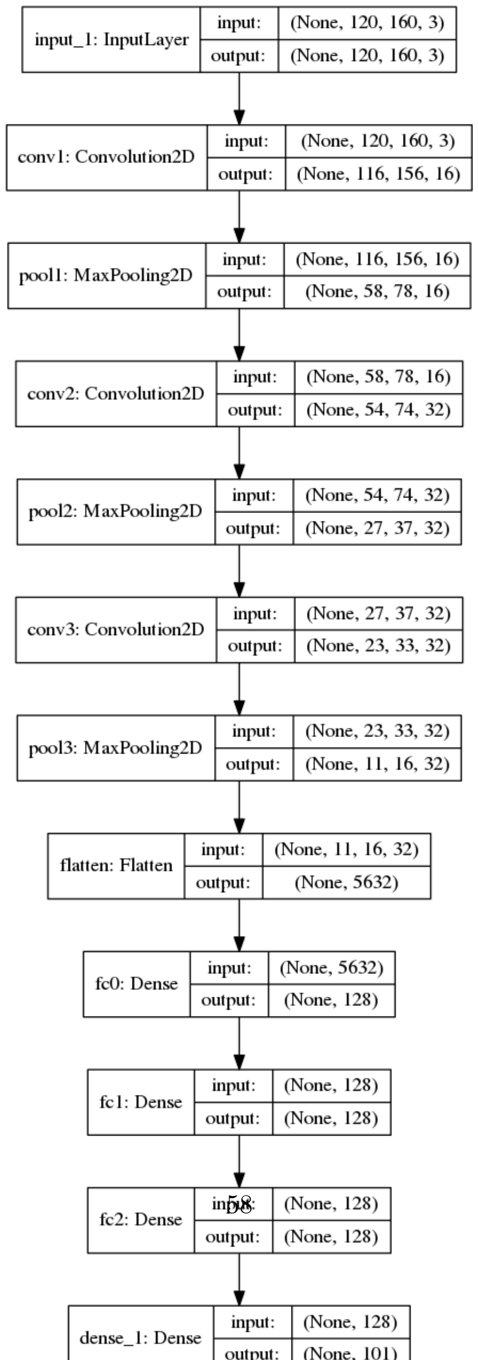
[42] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, 6:679–698, 1986.

[43] G. Bradski. Opencv. *Dr. Dobb's Journal of Software Tools*, 0, 2000.

Appendices

Appendix A

Models


input_1: InputLayer | input: (None, 120, 160, 3) | output: (None, 120, 160, 3)
conv1: Convolution2D | input: (None, 120, 160, 3) | output: (None, 116, 156, 16)
pool1: MaxPooling2D | input: (None, 116, 156, 16) | output: (None, 58, 78, 16)
conv2: Convolution2D | input: (None, 58, 78, 16) | output: (None, 54, 74, 32)
pool2: MaxPooling2D | input: (None, 54, 74, 32) | output: (None, 27, 37, 32)
conv3: Convolution2D | input: (None, 27, 37, 32) | output: (None, 23, 33, 32)
pool3: MaxPooling2D | input: (None, 23, 33, 32) | output: (None, 11, 16, 32)
flatten: Flatten | input: (None, 11, 16, 32) | output: (None, 5632)
fc0: Dense | input: (None, 5632) | output: (None, 128)
fc1: Dense | input: (None, 128) | output: (None, 128)
fc2: Dense | input: (None, 128) | output: (None, 128)
dense_1: Dense | input: (None, 128) | output: (None, 101)

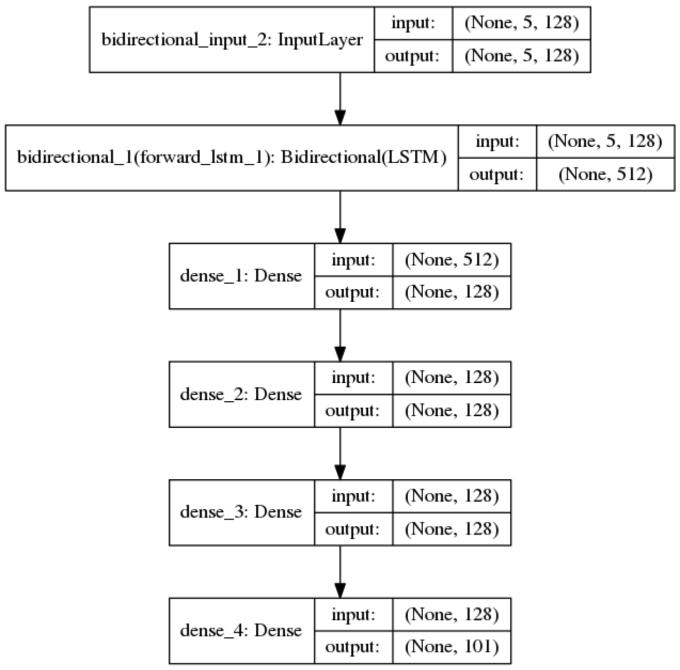


Figure A.2: Scenario 1, sequence of 5.

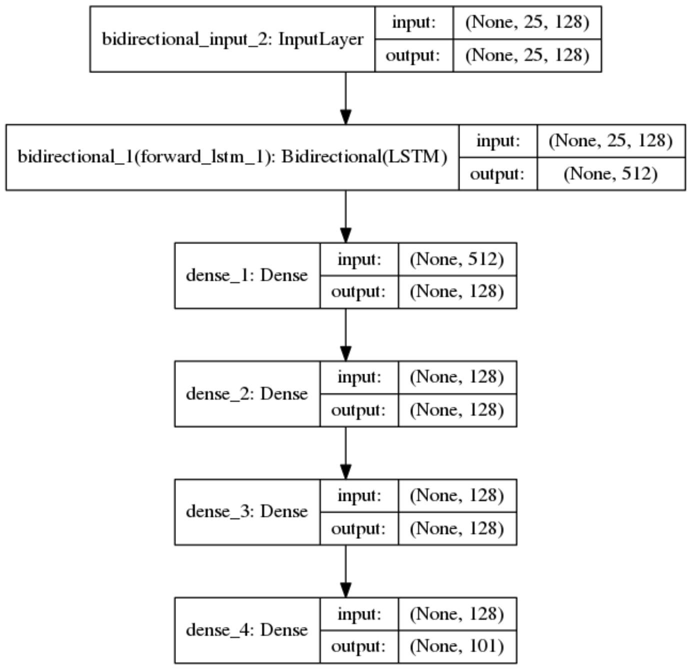


Figure A.3: Scenario 1, sequence of 25.

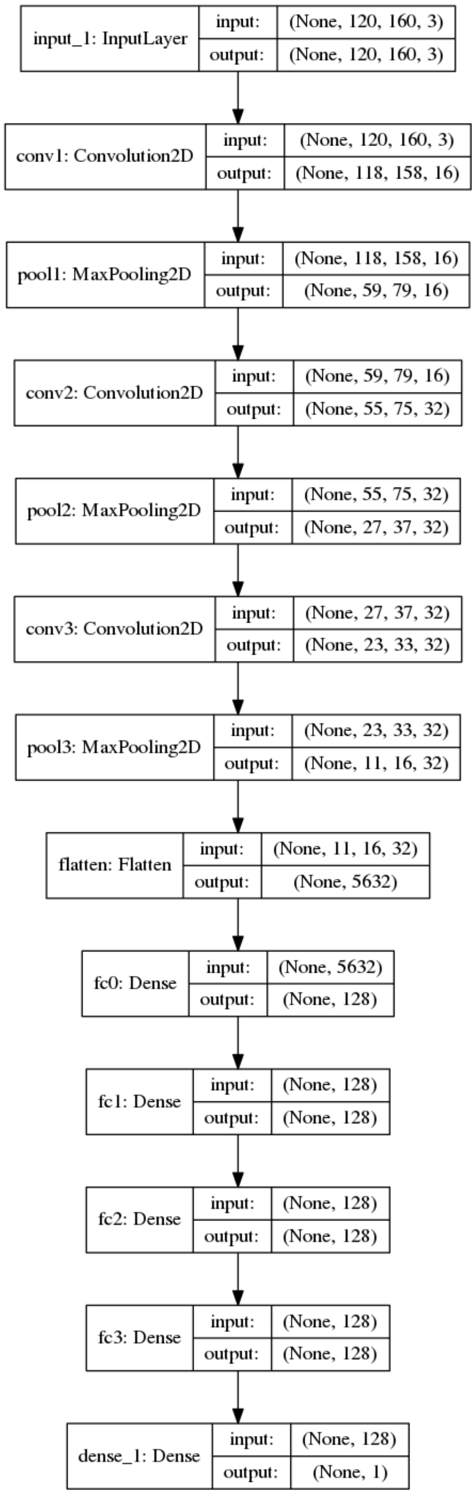


Figure A.4: Scenario 2, single-state.

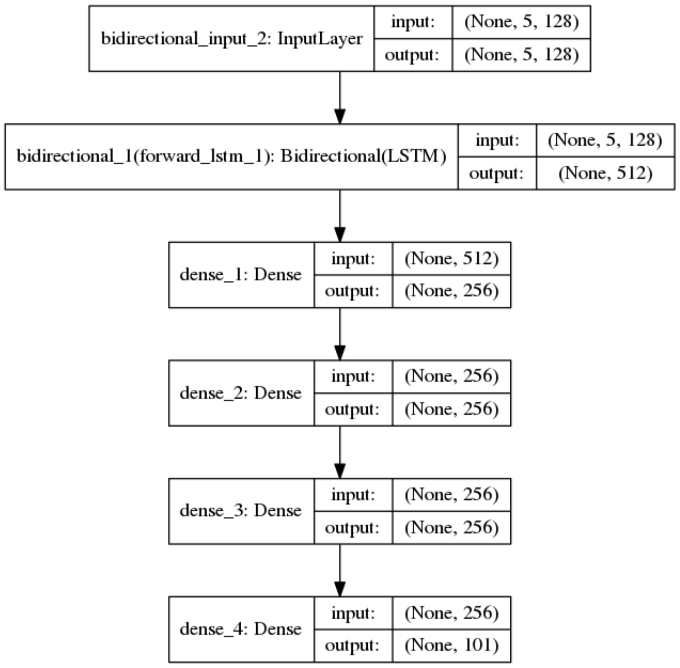


Figure A.5: Scenario 2, sequence of 5.

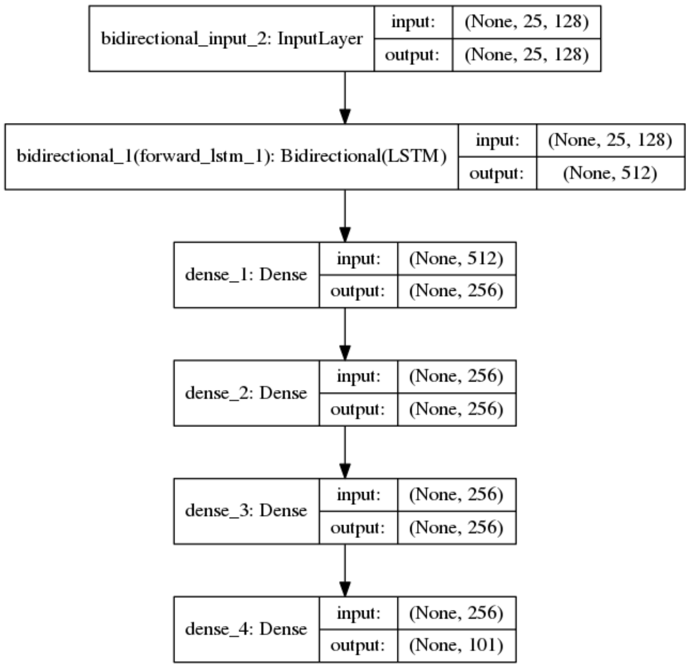


Figure A.6: Scenario 2, sequence of 25.

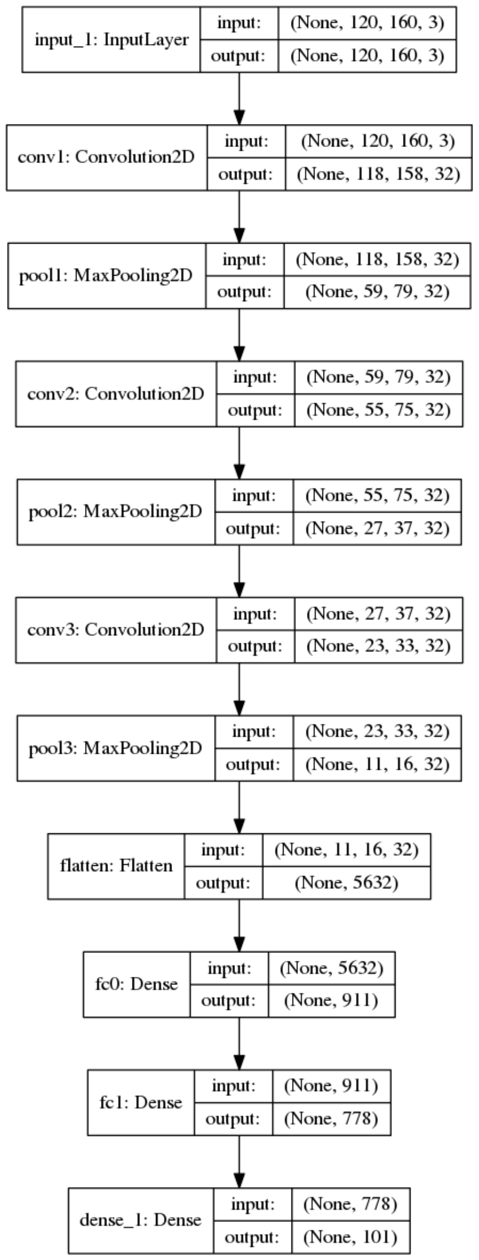

input_1: InputLayer
input: (None, 120, 160, 3)
output: (None, 120, 160, 3)
conv1: Convolution2D
input: (None, 120, 160, 3)
output: (None, 118, 158, 32)
pool1: MaxPooling2D
input: (None, 118, 158, 32)
output: (None, 59, 79, 32)
conv2: Convolution2D
input: (None, 59, 79, 32)
output: (None, 55, 75, 32)
pool2: MaxPooling2D
input: (None, 55, 75, 32)
output: (None, 27, 37, 32)
conv3: Convolution2D
input: (None, 27, 37, 32)
output: (None, 23, 33, 32)
pool3: MaxPooling2D
input: (None, 23, 33, 32)
output: (None, 11, 16, 32)
flatten: Flatten
input: (None, 11, 16, 32)
output: (None, 5632)
fc0: Dense
input: (None, 5632)
output: (None, 911)
fc1: Dense
input: (None, 911)
output: (None, 778)
dense_1: Dense
input: (None, 778)
output: (None, 101)

Figure A.7: Scenario 3, single-state.

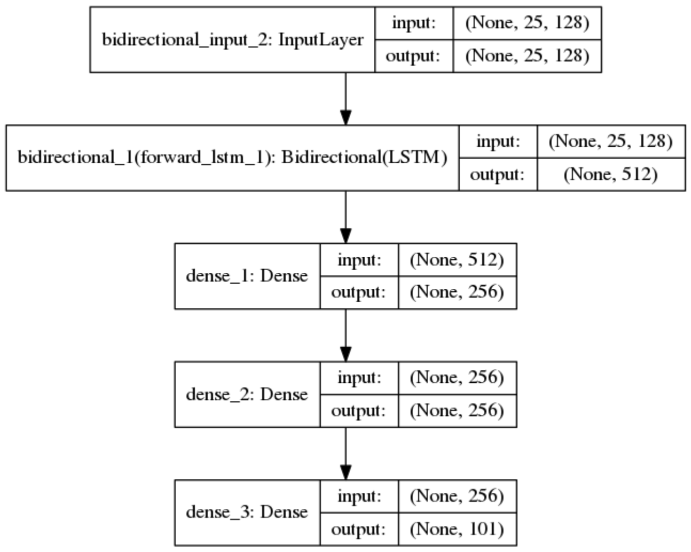


Figure A.8: Scenario 3, sequence of 5.

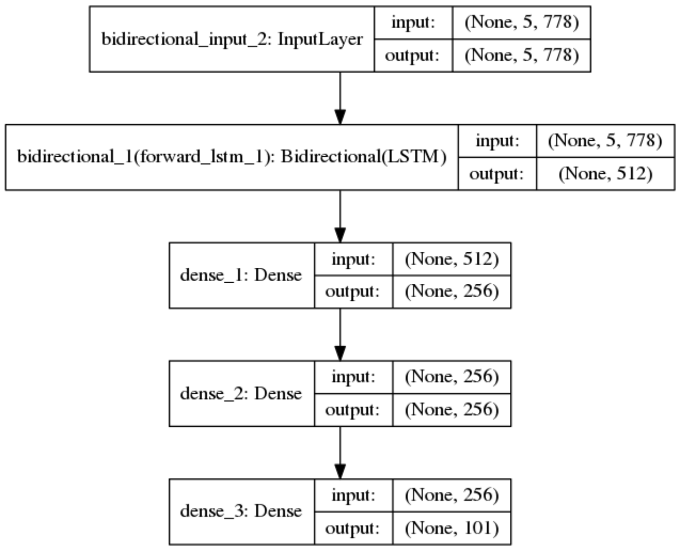
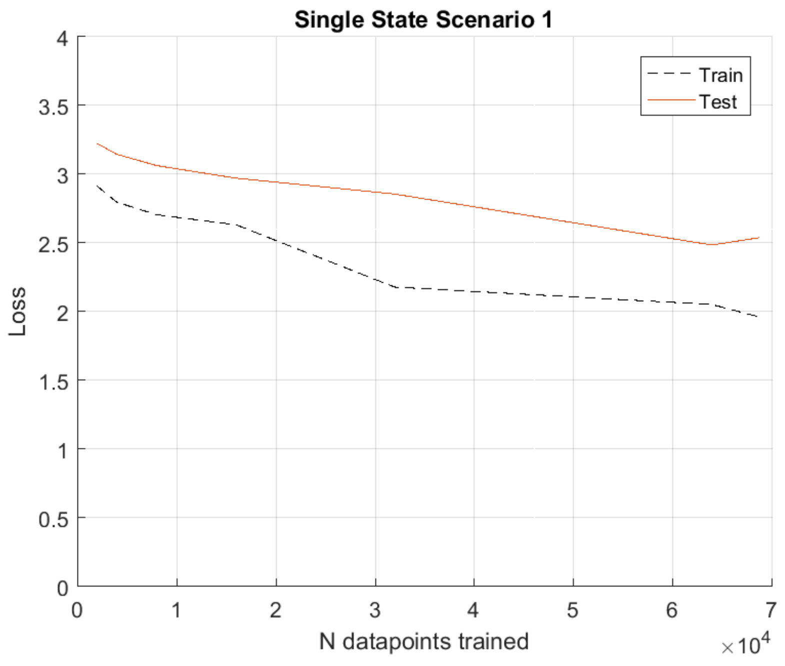
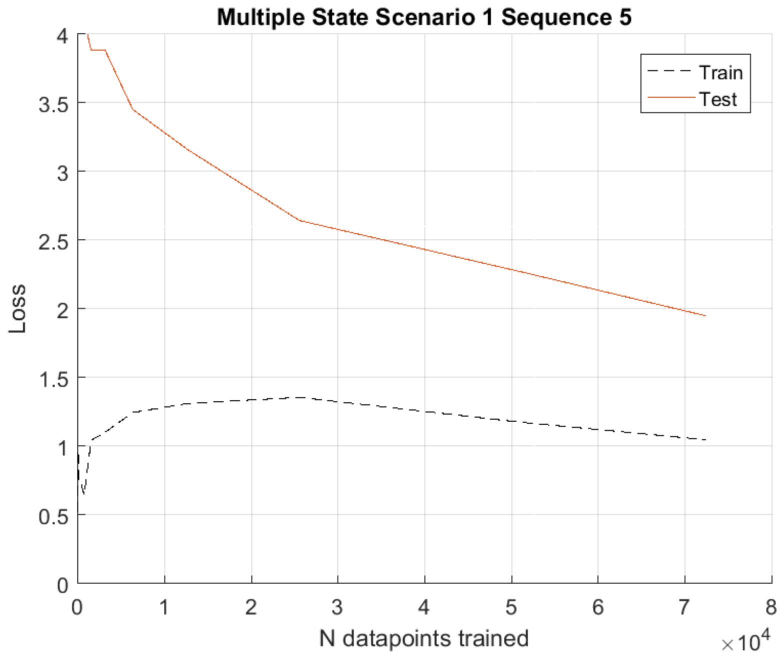


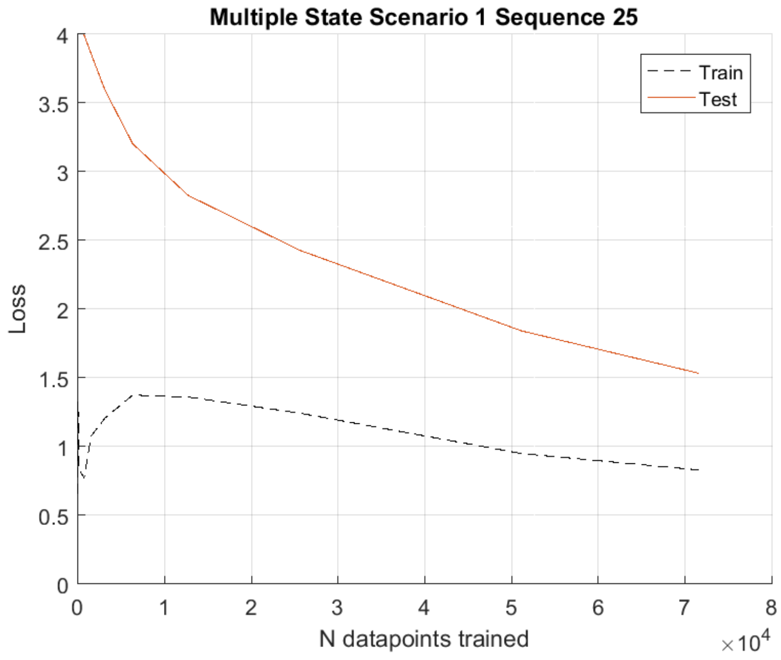
Figure A.9: Scenario 3, sequence of 25.

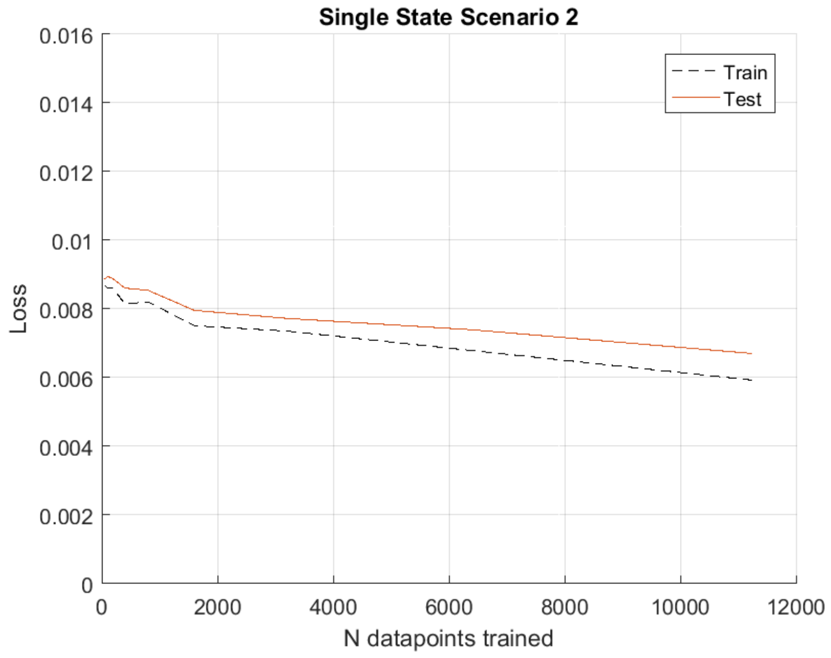
Appendix B

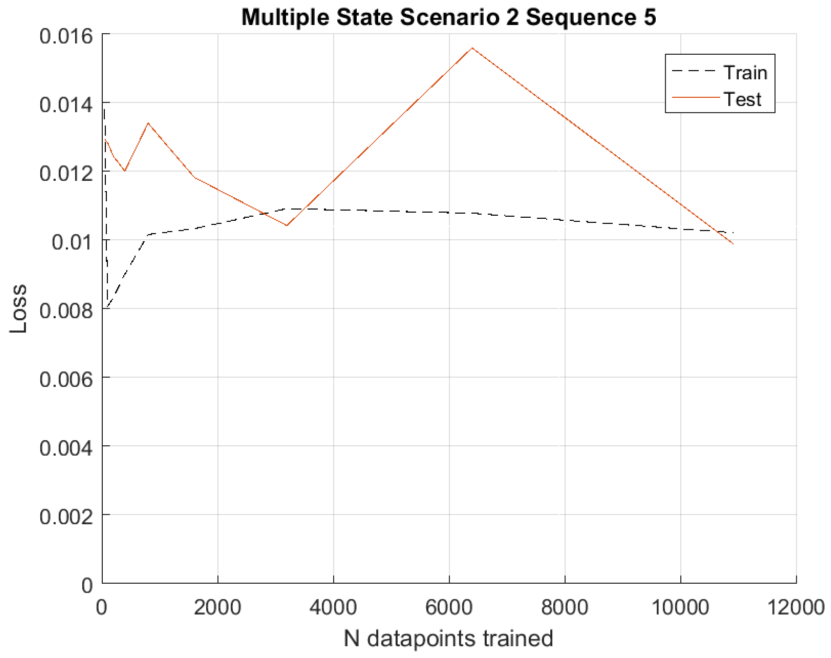
Learning Curves

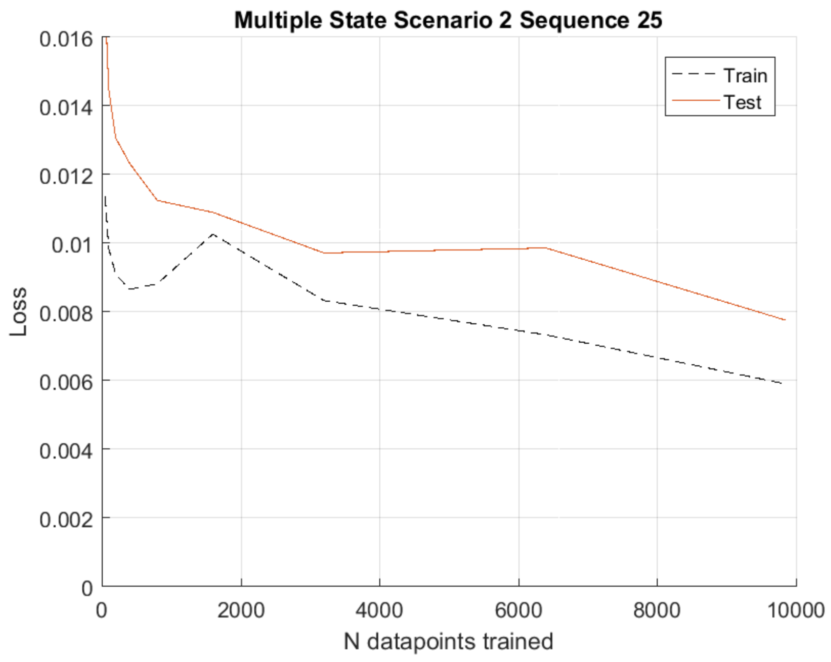


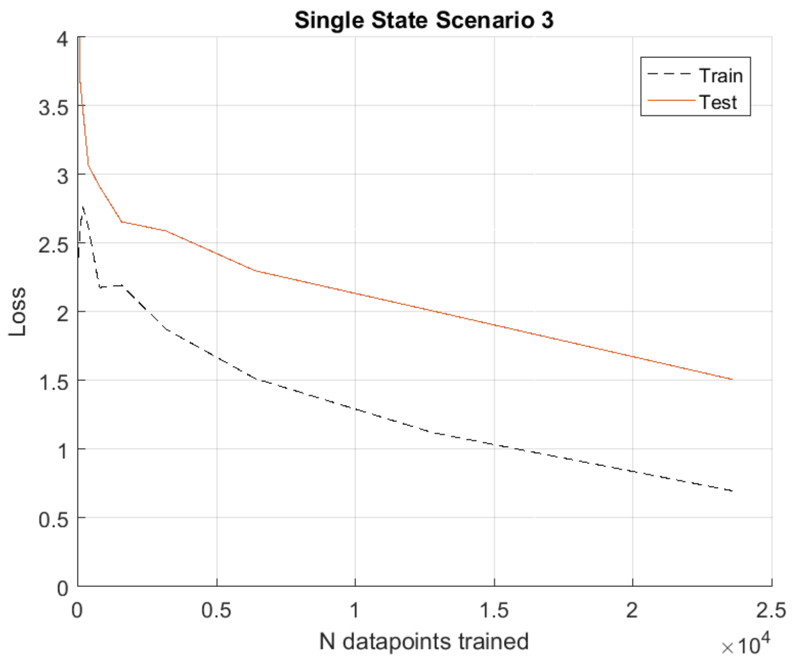

Multiple State Scenario 1 Sequence 5
Loss
N datapoints trained
$\times 10^4$
Train
Test
0
0.5
1
1.5
2
2.5
3
3.5
4
0
1
2
3
4
5
6
7
8

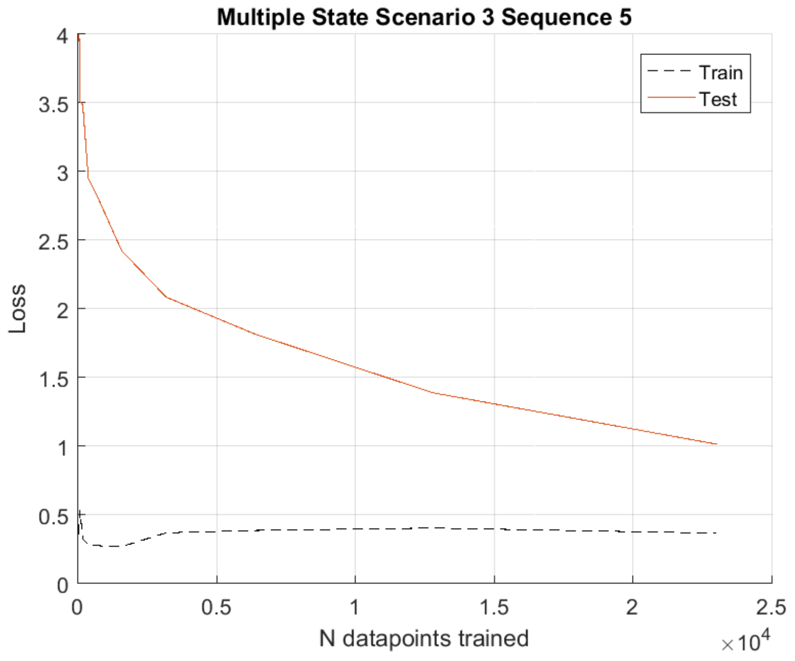

Multiple State Scenario 1 Sequence 25
Loss
N datapoints trained
$\times 10^4$
Train
Test
0
0.5
1
1.5
2
2.5
3
3.5
4
0
1
2
3
4
5
6
7
8

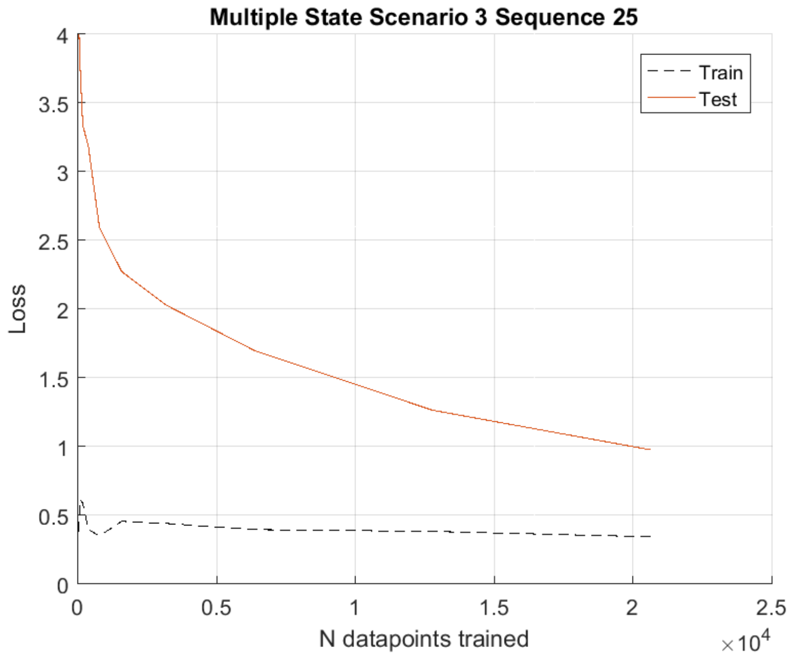

Single State Scenario 2
Train
Test
Loss
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
0
2000
4000
6000
8000
10000
12000
N datapoints trained


Multiple State Scenario 2 Sequence 5
Train
Test
Loss
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
0
2000
4000
6000
8000
10000
12000
N datapoints trained


Multiple State Scenario 2 Sequence 25
Train
Test
Loss
0
0.002
0.004
0.006
0.008
0.01
0.012
0.014
0.016
0
2000
4000
6000
8000
10000
N datapoints trained


Single State Scenario 3
Train
Test
Loss
0
0.5
1
1.5
2
2.5
3
3.5
4
0
0.5
1
1.5
2
2.5
N datapoints trained
×10^4


Multiple State Scenario 3 Sequence 5
Loss
N datapoints trained
×10^4
Train
Test
0
0.5
1
1.5
2
2.5
3
3.5
4
0
0.5
1
1.5
2
2.5


Multiple State Scenario 3 Sequence 25
Loss
N datapoints trained
×10^4
Train
Test
0
0.5
1
1.5
2
2.5
3
3.5
4
0
0.5
1
1.5
2
2.5

Appendix C

Division of Work

The work was very closely integrated and it is rather difficult to draw a line on who did what. A general split would be that Gabriel area of responsibility was the hardware, construction of the car and ROS - the communication between all components. Additionally he wrote theory, designed and trained the single-state network. Martin had responsibility over the Neural Networks, their training framework and theory, including the RNN.

- ROS - Gabriel
- Hardware design - Gabriel
- Hardware construction - Gabriel
- Hardware control software - Gabriel
- Single state network & Convolutional Neural Networks - Gabriel
- Artificial Neural Networks - Martin
- Recurrent Neural networks - Martin
- Keras - Martin
- Test cases - Martin