Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Jonas Dammen

End-to-end deep learning for autonomous driving

Master's thesis in Computer science
Supervisor: Frank Lindseth
July 2019

NTNU
Norwegian University of Science and Technology

Abstract

Deep learning has seen a recent advance due to more available data and increased computing power. This has lead to major advances in several fields as for example image classification and object detection. The advances have also made it into the automotive industry and companies like Tesla, Waymo, and Über are working hard at developing self-driving cars. They have achieved great results in the past years, but their progress is not publicly available. This thesis will contribute to the publicly available knowledge of self-driving cars. It explores end-to-end learning for autonomous driving in urban areas. Compared to lane-following, driving in urban areas introduce several challenges as various types of intersections, vehicles, and pedestrians. The task is complex and introduces challenges related to end-to-end architectures.

The paper *End-to-end driving via Conditional Imitation Learning* (F. Codevilla et al. 2018 [1]) proposed conditional imitation learning, where the network uses high-level commands about the experts' intention in the upcoming intersection. Expert intentions are necessary for an autonomous vehicle to operate in an urban area, and they proved that this solution solved a lot of the tasks related to driving in cities. During the writing of this paper, the simulator environment was limited in terms of variation of traffic elements, and thus it can only handle a subset of the obstacles encountered in urban areas.

This thesis proposes a stacked Convolutional neural network (CNN) that solves the task of driving autonomously using a timeseries of input, and compares it to a recurrent neural network and a simple CNN.

The solution was evaluated using different tracks in a town not seen during training, where the goal of the car is to get from one position to another without human interactions. During driving, different measurements are recorded to evaluate the performance of a given model. The results show that using a stacked CNN achieved 50% more tracks compared to a simple CNN, based on the same conditions and architectures.

Sammendrag

Dyp læring har sett stor progresjon på bakgrunn av mer tilgjengelige data og økt datakraft. Dette har ført til store fremskritt på flere felt som for eksempel bildeklassifisering og objektdeteksjon.

Fremskrittene har også funnet veien til bilindustrien, og selskaper som Tesla, Waymo og Über jobber hardt for å utvikle selvkjørende biler.

De har oppnådd gode resultater de siste årene, men deres fremgang er ikke offentlig tilgjengelig. Ved hjelp av denne oppgaven håper jeg å kunne bidra til mer offentlig tilgjengelig kunnskap om selvkjørende biler. Oppgaven undersøker ende-til-ende læring for autonom kjøring i byområder. Sammenlignet med landevei følger kjøring i byområder flere utfordringer som ulike typer kryss, kjøretøy og fotgjengere. Oppgaven er kompleks og introduserer utfordringer knyttet til ende-til-ende arkitekturer.

Artikkelen *End-to-end driving via Conditional Imitation Learning* (F. Codevilla et al. 2018 [1]) foreslår en metode kalt betinget etterligning, hvor et nevralt nettverket tar inn ekspertens intensjon om hvor man skal i det kommende krysset som input.

En overordnet intensjon er nødvendige for at et autonomt kjøretøy skal kunne operere i et byområde, og de viste at denne løsningen løste mange oppgaver knyttet til kjøring i byer. Under skrivingen av deres oppgave var simulatormiljøet begrenset når det gjaldt variasjon av trafikkelementer, og dermed kan det bare håndtere en delmengde av hindringene som oppstår i byområder.

Denne oppgaven foreslår et arkitektur som bruker en serie av Convolutional Neural Networks (CNN) som løser oppgaven med å kjøre autonomt ved hjelp av en tidsserie av input, og sammenligner den med et Reccurent neural network og med en normal CNN.

Løsningen ble evaluert ved hjelp av forskjellige baner i en by som ikke ble brukt under treningen, hvor målet er å komme fra en posisjon til en annen uten menneskelige interaskjon. Under kjøring registreres ulike målinger for å evaluere ytelsen til en gitt modell. Resultatene viser at den foreslåtte arkitekturen gjennomførte 50% flere baner enn den normale CNN modellen, basert på de samme forholdene og arkitekturene.

Preface

This thesis was written over the course of the spring semester 2019 for the Department of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU). It was written as a part of a larger project at NTNU that has as a goal to make a car self driven.

Acknowledgment

First of all, I would like to thank my supervisor Frank Lindseth for his supportiveness and guidance throughout the semester. I would also like to thank him for great ideas, and for suggesting to try out the architecture proposed in the thesis.

I would like to thank Max Aasboe and Hege Haavaldsen for great discussions, and also their willingness to help with problems encountered. At last I want to thank Ingvild Giset for helping with the finalization of the thesis, and for providing great feedback.

J.D.

Contents

Abstract . . . . . **v**
Sammendrag . . . . . **vi**
Preface . . . . . **vii**
Acknowledgment . . . . . **viii**
Contents . . . . . **ix**
List of Figures . . . . . **xi**
List of Tables . . . . . **xii**
1 Introduction **1**
1.1 Background and Motivation 1
1.2 Research Questions and Objectives 2
1.3 Contributions 2
2 Background **3**
2.1 Theory 3
2.1.1 Autonomous driving 3
2.1.2 Deep learning 5
2.1.3 Neural network architectures 8
2.1.4 Relevant techniques for deep learning 10
2.2 Relevant Tools 10
2.2.1 Carla (Car Learning to Act) 10
2.2.2 AirSim 12
2.3 Literature 12
3 Methodology **15**
3.1 Overview 15
3.2 Research plan 15
3.3 Tools and equipment 16
3.3.1 System setup 16
3.3.2 Simulator 17
3.4 Data gathering 17
3.4.1 Environment 17
3.4.2 Agent 17
3.5 Data preprocessing 20
3.5.1 Data preparation 20
3.5.2 Filtering 21
3.5.3 Data augmentation 21

3.5.4 Training data statistics . . . 21
3.6 Training . . . 25
3.6.1 Network architectures . . . 25
3.6.2 Callbacks . . . 28
3.6.3 Loss functions . . . 29
3.6.4 Training and validation set . . . 29
3.6.5 Optimizer . . . 29
3.6.6 Grid search . . . 29
3.7 Testing . . . 29
3.7.1 Setup . . . 29
3.7.2 Metrics . . . 31
3.8 Configuration used for final testing . . . 32
4 Results . . . 35
4.1 Training results . . . 35
4.2 Testing results . . . 37
5 Discussion . . . 40
5.1 Feature extractor and input size . . . 40
5.2 Dataset used for training . . . 41
5.2.1 Validation set to similar to training set . . . 41
5.2.2 Achieving consistent loss convergence with large amount of training data . . . 42
5.2.3 Garbage In Garbage Out . . . 42
5.2.4 Speed limit . . . 42
5.3 Comparison of architectures . . . 44
5.3.1 Test result . . . 44
5.3.2 Complexity of fitting a good model . . . 44
5.3.3 Model complexity . . . 45
5.3.4 Prediction time . . . 45
6 Conclusion . . . 47
6.1 Future Work . . . 47
Bibliography . . . 48
A Exploitation results . . . 49
A.1 Spatial . . . 51
A.1.1 Model 9 . . . 51
A.2 Spatiotemporal . . . 52
A.2.1 Model 13 . . . 52
A.3 Temporal . . . 54
A.3.1 Model 25 . . . 54
B Complete results from final testing . . . 56

List of Figures

1 Mediated perception 4
2 End-to-end flow 5
3 Training data statistics - before filtering 22
4 Steering distribution - before filtering 23
5 Training data statistics - after filtering 24
6 Steering distribution - after filtering 25
7 Spatial Architecture 26
8 Spatiotemporal Architecture 27
9 Temporal Architecture 28
10 Spawn-points - town 2 30
11 Training loss - Spatial model 35
12 Training loss - Spatiotemporal model 36
13 Training loss - Temporal model 37
14 Image size illustration - feature extraction 41
15 Autopilot crashing high speed 43
16 Autopilot crashing 44
17 Spatial model 9 51
18 Spatiotemporal model 13 52
19 Temporal model 25 54
20 Temporal model 25 - Exploitation test results 55

List of Tables

1 System setup 16
2 Recorded measurements 20
3 Final testing tracks 31
4 Metrics for testing 32
5 Final configuration of the spatial model 32
6 Final spatiotemporal training configuration 33
7 Final temporal training configuration 34
8 Overall statistics from the testing based on network architecture . . . 37
9 Targets reached based on driving with or without other cars 38
10 Targets reached based on weather conditions and network architectures 38
11 Targets reached based on track and network architecture. 39

1 Introduction

1.1 Background and Motivation

Deep learning has recently been widely adopted in the field of autonomous driving. The topic has been explored for several years, and we see some older results as ALVINN presented by D.A. Pomerleau et. al. [2] which showed promising results in terms of lane following. That being said, it is after the deep learning revolution, triggered by the creation of Imagenet, that deep learning has been adopted to several tasks related to autonomous driving. Unfortunately, there is not a lot of publicly available data on the topic, compared to the state of the art achieved by companies like Über, Waymo and Tesla. This might be a result from the difficulty of gathering data and testing models for those who are not car manufactures. Fortunately, there has been a recent advancement in available simulators, which enables more people to explore the topic.

The task of driving in urban areas introduces a lot of problems compared to the task of lane-following. The task of lane-following can be solved by a CNN that uses only an image as input, and then outputs the degree of steering. In an urban area, the task gets more complex and a single image doesn't provide enough information for the model to predict the next action. The task introduces interaction with other cars, which creates the need for temporal knowledge. It introduces the need for path-intention in intersections. Last but not least it introduces law enforcement in the way of road signs that provide the vehicle with information like speed limit, and traffic lights which tells you to drive or wait.

The paper *End-to-end driving via Conditional Imitation Learning* (F. Codevilla et al. 2018) [1] proposed conditional imitation learning, where the network uses high-level commands about the experts' intention in the upcoming intersection. Expert intentions are necessary for an autonomous vehicle to operate in an urban area, and they proved that their solution solved a lot of the tasks related to driving in urban areas. During the writing of their paper, the simulator environment was limited in terms of variation of traffic elements, and it can only handle a subset of the obstacles encountered in urban areas. Another limitation is that the networks top layers are divided into several branches. Each branch is only trained on a subset of the scenarios, based on the expert intention. Thus, one branch has for example only been exposed to right turns during training, and therefore only learned scenarios that have happened in a right turn. This introduces problems when the model has to handle rare cases as a pedestrian walking into the road.

1.2 Research Questions and Objectives

The objective of this thesis is to explore new architectures for driving in urban areas, and to do a comperative study between models that only use spatial information and models which includes temporal information. More formally this thesis will look into three research questions.

Research question 1

Recurrent neural networks (RNN) are usually used for tasks that are concerned with temporal information. These are however hard to fit, due to their complexity. As an alternative you can stack several Convolutional neural networks (CNN), where each CNN takes in an image from a given timestep. How will such an architecture compare to a RNN architecture? Which benefits or limitations do each provide in terms of autonomous driving?

Research question 2

How does a temporal model compare to a spatial one, is it possible to drive well using only spatial information, or is temporal information a necessity for safe driving.

Research question 3

F. Codevilla et. al. [1] writes in their discussion:

> "While the presented results are encouraging, they also reveal that significant room for progress remains. In particular, more sophisticated and higher-capacity architectures along with larger datasets will be necessary to support autonomous urban driving on a large scale. We hope that the presented approach to making driving policies more controllable will prove useful in such deployment."

Given the knowledge that a model with high capacity will easily overfit the data if not enough variance in the training samples is provided. How does the capacity of the model compare to the generalization of the model? Can a model with less capacity achieve better results compared to one with high capacity?

1.3 Contributions

This thesis proposes an architecture that uses expert intention as regular input, thus avoiding condition based branching. It also adds temporal dependencies to achieve good driving with other actors in the environment, this is done with and without a recurrent layer, where the best results was achieved without a recurrent layer.

2 Background

This chapter consists of a theoretical background, an outline of the relevant tools and a historical background. The theory section will cover relevant topics for the thesis, it will start broad then get more technical. The relevant tool section will provide information about the simulator used for the project and one alternative that was considered. The historical background is structured like a topologically sorted literature review. It serves as a purpose to outline how we have gotten to the current state of the art.

2.1 Theory

2.1.1 Autonomous driving

C. Chen et. al. 2015 [3], outlines two major paradigms for vision based autonomous driving and proposes a new one that falls in between the two paradigms. The first paradigm is Mediated perception, which involves sub-components for creating a representation of the surrounding environment. The second paradigm is behavior reflex, which directly maps sensory input to an output action using a regressor. Behavioral reflex is more widely known as end-to-end learning, which is the term that will be used from now on. The approach proposed by C. Chen et. al. [3] is called direct perception and compared to the behavioral reflex it maps the input image to affordance, and then uses a simple controller to compute the actions. For this thesis, the two main paradigms are of most interest, and they are also the ones that will be further described. Each of the approaches have their different benefits and limitations, and having a basic understanding of the limitations of each approach is important before starting to develop self driving cars.

Mediated perception

Mediated perception creates an understanding of the surrounding environment by leveraging information gained from several components. Each component has a subtask of recognising relevant information as for example traffic lights, pedestrians or other cars. The information provided by each component is combined to create a complete understanding of the environment. Then an agent uses this information to compute an action. This approach combines computer vision, sensor fusion, localization, control theory, and path planning to get the desired information about the environment.

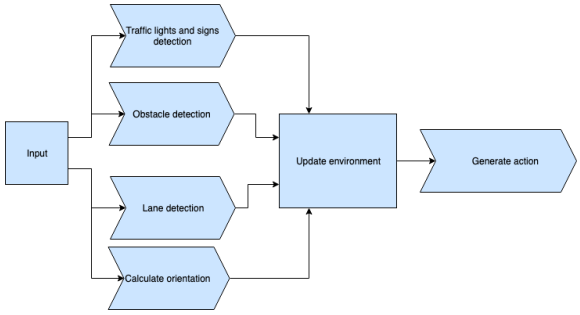


Figure 1: Example of the different modules in a mediated perception

Figure 1 provides an example of mediated perception. The environment can be provided as a three-dimensional map that has been recorded beforehand or generated while driving using for example Lidar and SLAM. During driving, several sub-processes generate information that is important to the driving, as for example detection of obstacles in the road. The information is then aggregated and the state of the environment is updated. By using path planning and the environment state the agent can set a goal for the car. The last step is to use a controller to generate action based on the state of the environment and the current goals.

Since the feature extraction is done by sub-tasks, it's possible to test each sub-task by itself to verify that it works as expected. It's therefore easier to tell why the car takes a certain action, compared to a end-to-end approach, where the reasoning is more or less hidden in a black box. The mediate perception approach often involves creating a mapping of the environment using for example a Lidar. Cars adopting this approach therefore relies on driving in previously mapped areas, and the autonomous driving is limited elsewhere.

End-to-end learning

End-to-end learning is concerned with developing a network that can go from as raw data as possible to the final-most output as in figure 2. One of the main goals is to achieve as little as possible engineering to the input or output, and instead let the network learn to do the job of finding out how it should act based on a given input. This methodology has gained a lot of tension, especially in the autonomous driving industry, following the growth of deep learning. The network will do the full job of feature extraction, detection, classification and regression to get the desired output.

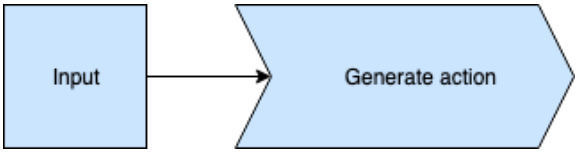


Figure 2: Example of the simplicity of an end to end network in comparison to mediated perception as seen in figure 1

The network does npt hold any information about the state of the environment and most therefore deduct this from the input during inference. Compared to mediated perception it is less sophisticated and relies heavily on the input provided. The function that maps the input to the output should therefore be deterministic. This means that given an input, there should not be several optimal goals. As an example, Yann LeCun et. al. 2005 [**?**] encountered this when training a robot to avoid obstacles. It could drive either to the right or left when an obstacle was encountered, and both would in theory let the robot avoid the obstacle. During training however, only one option is the correct one, and this is based on the drivers intuition at the moment of driving. This intuition is not provided to the network and the robot will have no way to know which way is the correct one, and will therefore struggle to learn a deterministic mapping.

The benefit of the end-to-end approach is that the network is able to adapt fast to new conditions in the environment. Once a sufficient model is developed, it should be enough to expose the model to training data that contains these conditions, and the network will learn itself how to handle them. Since it doesn't rely on Lidar to create a mapping of the environment, a big cost is also removed from autonomous cars.

A disadvantage compared to mediated perception is that it can't think ahead. It works more like intuition, and will therefore need a system that can do more complicated reasoning. For example, an end-to-end model can learn to brake, steer and accelerate based on the input, but since it doesn't hold any information about the environment, it can't create an alternative route and then follow it if something is blocking the road.

2.1.2 Deep learning

Neural networks

Artificial neuron(AN) and Artificial neural networks(ANN) are biologically inspired from neurons in the brain and try to create a mapping between input and output, much similar to how your brain reacts (output) to something it sees (input). ANN learns this mapping by combining a forward pass and a backward pass over several input-output pairs.

A feedforward neural network(FNN) is the basis of deep learning models. The FNN is trying to optimize a function f that maps the input X to an output y. It's called feedforward because information flows from the input X, and forward through the network until it reaches the output y. A typical representation of a feedforward network can be given by the formula

$$f(x) = f_1(f_2(...f_{n-1}(f_n(x))))$$

where f_i is a layer in the network consisting of n layers.

Each layer in the network consists of a set of nodes that are trying to represent the function of a neuron. A node receives inputs from the output of the nodes at the previous layers. This output is multiplied by the weights of the network and a bias is added. Thus for a simplified FNN, we will get the following formula for the output of each node j in layer n:

$$f_{n,i}(x) = W_{n,i} * x + b_n$$

where b is the bias.

The weights, W, are the parameters that we attempt to learn so that we can approximate the target function.

The node described above has a linear relationship between the input and the output and can be useful for linear regression. In most scenarios where deep learning is applied, we are trying to describe a nonlinear function and thus we will need to add activation functions that add non-linearity.

Activation functions

The activation function takes a set of input values and maps it to a value in a range given by the function chosen. The output is then used as input for the next node in the network. There are several activation functions to choose from and the ones used for this project is described here.

The sigmoid function

The sigmoid function maps the input to the range [0,1] for each node, by applying the function $f = 1/(1 + e^x)$. This function can be thought of as a node that outputs the probability that it should be activated independent of the other nodes.

Sigmoid function has been one of the most widely used activation functions and is doing great for classification problems. Unfortunately, the sigmoid function suffers from the vanishing gradient problem, which comes from the characteristics of the function. When the activation approaches the horizontal part of the curve, the gradient will be very small and the network will struggle to learn.

The Relu function

Relu looks similar to the linear function but the tail ($x < 0$) is different. The function is given by $f(x) = max(0, x)$, and has a range from $[0, inf]$, which means that the activation can blow up. The Relu function is less computational expensive but combinations of Relu functions can still approximate every function. The Relu function has grown a lot in popularity since it was demonstrated in 2011, and is today widely used for deep learning.

Exponential linear unit (ELU)

ELU is quite similar to the linear rectified unit, but it has the function $f(x) = a(e^x - 1)$ when x is less than 0. The range is from $[-a, inf]$.

Loss function

The loss function (also called cost function) is a function that gives you a measure on how good the prediction is at predicting the expected outcome. It does this by comparing the output of the network with the expected output, using different functions for comparison. There isn't a single loss function that works better than others, and thus you have to choose the function depending on the problem you are trying to solve.

Mean square error (MSE)

For regression, one of the most popular loss function out there is MSE, it works by computing the sum of squared distance between the predicted value and the expected value. The MSE function works well but is heavily affected by outliers, this has to do with the square of the error. So when the error grows larger than 1, then the MSE will grow exponentially.

Binary cross entropy

When doing classification, it is more suitable to use cross entropy. It measures performance based on output being a probability of something being correct or not. Binary cross entropy is used when the output of the model only can be true or false, and is given by the following formula:

$$loss = -(y * log(p) + (1 - y)log(1 - p)).$$

Optimization algorithm

During backpropagation in a network, the weights are updated using an optimization algorithm. Gradient descent is the most commonly known optimization algorithm in deep learning and it works by minimizing a loss function J(θ), where θ is the model parameters. To minimize the loss, it updates the parameters in the opposite way of the gradient of the loss function at each step during training. To define how much the weights should be updated during each update one multi-

plies a learning rate with the gradient. The formula for a gradient descent update is given by:

$$\theta = \theta - n\frac{\partial J}{\partial \theta},$$

where θ is the parameters to be trained, n is the learning rate and J is the loss. Gradient descent comes in three variants, where the first one is batch gradient descent. This variant computes the gradient of the loss using the full training set before it updates the model parameters. It is usually slow and it's a necessity to keep the full training set in memory, which is often not possible during deep learning. The second variant is stochastic gradient descent which updates the weights after each training sample. This causes faster training, but also a lot of fluctuation. The third variant which is the one that is most adapted is batch gradient descent. Batch gradient descent computes the gradient for a batch of training samples. Thus, only a smaller subset of the training data needs to be in memory at the same time and the fluctuation from updating after every sample is reduced.

Gradient descent has proven itself to work very well, but it is often hard and time consuming to find the correct learning rate. The learning rate needs to be large enough to converge at a tolerable speed, but it can't be so large that it causes fluctuation, and thereby prevents any learning from happening. Finding this middle point is hard and takes a lot of time. During gradient descent, it's often easy to end up in a local minima (sub-optimal state), and it might be hard to get out using gradient descent. To solve this problem Adam and RMSprop has been developed.

The RMSprop optimization algorithm is an algorithm that divides the gradient by a running average of its recent magnitude and it's usually a good optimizer for recurrent neural networks.

Adam is compared to gradient descent an algorithm that computes adaptive learning rates. It works well in practice and is faster than the gradient descent.

2.1.3 Neural network architectures

Spatial vs Temporal information

For the task of urban driving, the difference between spatial and temporal information has a lot to say. When the car drives around without any other actors, it is sufficient to use a spatial model like CNN. When other actors are introduced, it is not sufficient to use a spatial model. For example, if the model only takes in one image per timestep and it sees a car in front of itself. How does it know if the car is moving at the same speed as itself, or if the car is standing still due to queue? It can't know, and therefore need more information. A temporal model, would in comparison use images from a given amount of timesteps, and therefore

have enough information to tell if the car is approaching you or not.

Convolutional neural network

A convolutional neural network is a neural network that uses convolutional layers. This makes it very good at detecting features in the input, and it's widely used for tasks involving image classification or detection.

A convolutional layer is a layer that uses filters(kernels) to perform a linear operation on the input and is given by the following formula for two dimensional data:

$$S(i,j) = \sum\sum I(i-m, j-n)K(m,n),$$

where S is the convolution, I is the image, K is the kernel and m, n defines the size of the kernel. In a convolutional layer, the parameters to be learned is the kernel itself. That same kernel is used on the whole image, thus making the convolution equivariant to translations in the input.

One of the advantages of CNN is that it learns the filters (kernels) to use, thus it needs relatively little preprocessing compared to other image-classifiers, where features need to be defined beforehand.

Recurrent neural networks

Recurrent neural networks are neural networks that pay respect to the time series. This means that it is very useful for video, text, music and similar inputs where a given prediction not necessarily is dependent only on the current input but also on the input from the t last timesteps.

A simple recurrent layer receives the input xt, and the state v passed from the hidden layer at time $t-1$. The problem with this, is that it's biased against recent events in the input and thus it struggles with long-time memory. An architecture that handles long term memory better is the Long Short-Time Memory (LSTM) layer. LSTM uses forget gates to avoid the bias against short time memory and thus it avoids the bias against recent events.

The most common architecture of a LSTM unit consists of a memory cell, an input gate, an output gate and a forget gate. A given LSTM unit takes as input the previous cell state $Ct-1$, the hidden state vector from the previous timestep (often the output) $ht-1$, and the input vector xt. It then performs a series of concatenations (merging of two vectors), activations and multiplications. The forget gates calculate how much of the cell state should be passed along, and how much of the input that should be passed along. It does this by multiplying the probabilities with the input and the cell state.

2.1.4 Relevant techniques for deep learning

Regularisation

Regularisation is a technique used to punish too complex models and often works towards the purpose of better generalization. Regularisation is important to avoid overfitting the estimated function too well. Overfitting is when you have a mapping that corresponds too well to a given data, and is thus not able to produce the correct output for data that differs from the given dataset. An example is if you are trying to fit a multinomial function of a high degree to a polynomial function. The opposite would be to try to fit a linear function to a polynomial function and is called underfitting.

Overfitting is very relevant for this project since the networks are able to learn very complex mappings if trained long enough. The challenge with DNN is that there are so many parameters in the model, and without care, they can learn to memorize the training data directly instead of creating general "rules" for prediction.

There are several approaches to regularize, earlier in the report we talked about data augmentation and how you can augment the data to better generalize. Here we will provide some techniques used for regularisation of the network.

Dropout

An effective technique used a lot in RNN and CNN. A dropout layer will discard a given percentage of the neurons in a layer at the forward pass. Which neurons it discards is random at each iteration thus a neuron can't rely on a given input, and the model is forced to be more generalized.

Early stopping

Early stopping prevents the network from overfitting by stopping the training when the performance on a validation set declines or doesn't get any progress. To use early stopping you need to split the data into a training and validation set.

2.2 Relevant Tools

2.2.1 Carla (Car Learning to Act)

Carla is an open-source simulator for autonomous driving research. The simulator, released in 2017, was first mentioned in the paper *CARLA: An Open Urban Driving Simulator* (Alexey Dosovitkiy et. al. 2017) [4]. The goal of the simulator is to support the development, training, and validation of autonomous urban driving systems. The developers discovered that the current simulation platforms at that time were limited. Other open source simulators as TORCS [5], didn't include the complexity of urban driving, while games that simulate urban environment well, as Grand theft auto V, didn't support proper benchmarking techniques.

Carla is built on top of Unreal Engine 4 and provides state of the art rendering quality with realistic physics. It is built with a server-client architecture, where the server runs the simulation and the autonomous agent is the client. A python API is provided on the client side, enabling the user to choose an alternative to C++.

When published in 2017, the simulator had two towns to drive in. They had European traffic lights, speed limits ranging from 30-90km/h and other actors as vehicles and pedestrians. Since then, they have released 5 new towns, with several new elements, as for example highways, multi-lane intersections, and roundabouts. They also have a wider variety of actors, like bicyclists, motorcyclists and cars. The simulation environment can also be configured with up to 14 different weather scenarios, including heavy rain and sunsets, which creates realistic training environments.

The simulator is currently in their alpha stage with their newest release being 0.9.5 at the time of writing. The result of such an early stage simulator is that there are some bugs/unexpected behaviors in the python API, and testing of expected behavior is thus important when using Carla.

The server

The server runs the simulation environment, and provides information to the vehicle controller. It can be configured in many ways, suiting the need of the user. The server has a benchmark mode, which enables the user to define the amount of frames that should be rendered each second. It does this by slowing down the simulation time, so that a second in the simulator might lasts for five seconds in real time. This is a useful feature to achieve consistency during testing, and to be able to test more complex models that needs more time to predict.

The client

The Carla simulator is integrated with a python API that includes several agents. The Basic Agent is configured to drive from a start position to a goal position using a PID controller. First the agent finds the fastest route using A*-algorithm. Then, it uses a local planner to generate the next steps at a given time. These steps is provided as waypoints, which the PID controller uses to calculate steering. The throttle is calculated based on target speed and current speed, where target speed is the speed limit. The PID controller is only based on the current and previous error, and thus uses little temporal information when calculating the next action. It also uses a differential term that is hardcoded to 20fps.

The brake-command is based on hazards in front of the car. It is either triggered by a traffic light being red or if a car in front is too close. If one of these hazards occur, it will trigger an emergency stop signal. The emergency stop signal sets all controller signals to zero except for brake, which is set to one.

2.2.2 AirSim

AirSim is a simulator for autonomous vehicles also built on top of Unreal Engine 4. It is developed by Microsoft and supports a lot of the same features as Carla. The simulator was originally developed for research on autonomous drones, but added support for cars in 2017. They provide APIs that can be used in languages like C++ and Python, and as Carla, they provide several examples for getting started with for example end-to-end learning.

In many ways, AirSim seems to be quite similar to Carla in terms of providing a simulator suitable for AI research. There is one big difference though in terms of development setup. AirSim is built for Windows, thus there are only binaries available for Windows. That being said, it is possible to build it for Linux, but the only tested environment to date is ubuntu 16.04 LTS.

2.3 Literature

This section will explore the progress of autonomous driving, and outline recent approaches. It will be outlined in chronological order, and the purpose is for the reader to gain knowledge of the history of autonomous driving, including the recent findings.

ALVINN: An autonomous land vehicle in a neural network (1989)

One of the early attempts on autonomous driving using an end-to-end approach was ALVINN as presented in the paper by D.A. Pomerleau et. al. [2]. They proposed a neural network consisting of three layers to learn the task of road following. The model took two images as input, one of the road in front with a resolution of 32x32 pixels. The second image was from a laser range finder and consisted of 8x32 pixels. As the output layer, they had 45 units that function similarly to a softmax layer where each unit was zero except a hill of activation around the unit representing the correct turn. Each unit was related to a steering degree, where the unit in the middle corresponded to straight driving. They trained on a set of 1200 simulated images for an amount of 40 epochs. The results were that the network correctly classified a turn curvature within two units of the correct answer on approximately 90% of the time on simulated images. The testing was done on a 400-meter road in a wooded area where the car was able to drive at half a meter per second. The results were compared to the work of the vision and autonomous navigation groups at CMU. They proved that a network using backpropagation could learn in half an hour what groups at CMU used many months on. In their discussion, they also explain the need for local connectivity for the network to use knowledge about the two-dimensional nature of a picture.

Off-Road Obstacle Avoidance through End-to-End Learning (2005)

Following on the work of D.A. Pomerleau et. al. 1989, Yann LeCun et. al. [6] proposed a new architecture for end-to-end learning. In the period from 1989 to 1994, Yann LeCun developed the first Convolutional Neural Network called LeNet, which saw its first big application area in digit-classification on bank checks. This architecture was later used for off-road obstacle avoidance. Compared to ALVINN there were three major differences in their approach. First, the system used stereo cameras. Second, they trained for off-road obstacle avoidance, compared to road-following. Third, they used CNN instead of a Fully connected neural network.

They used two measures when testing their model, namely loss(MSE) and accuracy. The accuracy was measured by comparing the predicted steering angle, quantized into three bins(left, right and straight). During testing, they had an error rate of 35,8% which seems to be relatively high. But they encounter a problem which was the complexity of multiple viable choices. When the vehicle approached an obstacle it could either turn left or right. But only one of those would be classified as correct. This problem of multiple choices has been further investigated by F. Codevilla et al. in 2017 with their approach, called conditional imitation learning, which will be described more thoroughly later in this section.

NVIDIA - End to End Learning for Self-Driving Cars (2016)

The paper *End to End Learning for Self-Driving Cars* [7] published in April 2016 shows how a vehicle can follow lanes and roads without the need for engineers to tell an algorithm what to look for in an image. Invidia trained a convolutional neural network for lane following, compared to the work of D.A Promeleau et. al. 1989 [2], they now had a network that could capture local connectivity. They were also able to gather real-life data, by driving around in central New Jersey and from Highways from some other cities. They collected 3,000 miles of driving, with a sampling rate of 10 FPS and augmented the data with rotation and shifting.

The architecture is a straightforward CNN consisting of a normalization layer, six convolutional layers, and four dense layers. The network thus has 250 000 parameters to be trained. The testing of the network was done first in simulation, and when they got good enough results there, they tested the model on real roads. They measured performance by how long time the car could drive without interceptions and they achieved 98% when driving in Monmouth County and 100% when driving 10 miles on the Garden state parkway(highway). They showed that simple CNN's could learn the entire task of lane and road following, without the need for manual decomposition of features.

The previously mentioned papers looked at the task of lateral vehicle control, using steering as output. They have proven that the task of lane-following is solvable

with a relatively small neural network. But new problems arise when end-to-end learning need to predict longitudinal vehicle control as well as later vehicle control. The complexity increases since lane markings no longer are enough to predict the correct output.

End-to-end Driving via Conditional Imitation Learning (2017)

A more recent paper by F. Codevilla et. al. [1] published in 2017 addresses the problem that end-to-end systems cannot be controlled at test time. The problem arises when the driving task includes intersections, and the trained agent needs to know which way to take. The camera-input is no longer sufficient to decide whether the car should turn left, right, or go straight. To solve this issue, they proposed conditional imitation learning, where the network uses high-level commands about the experts' intention in the upcoming intersection. Based on the experts' intention the neural network switches between 5 different branches; Each one covering a specific scenario, as for example right turn in an intersection. This approach worked well in urban areas and it successfully completed 88% of the episodes in Town 1 and 64% in Town 2, compared to the same approach without experts' intention, which completed 20% of the episodes in Town 1 and 24% in Town 2.
One of the limitations with this model is that each branch will only be trained on a subset of the scenarios available in the training set. For autonomous driving, where there are a lot of cases that happen rarely, this might be critical. Take the example of a pedestrian crossing the road. During training, the car has seen several samples where a pedestrian crosses the road. But all of these have been during either a right or a left turn. During testing, the car comes across a pedestrian crossing the road during an intersection, but the car is now supposed to drive straight. A branch that hasn't been exposed to training examples where pedestrians are crossing is now making the prediction and might be unaware of the correct decision to make.

In their paper, they don't mention how their model handled scenarios where temporal information is a dependency. When driving on a road without cars, CNN should be able to drive quite well. When other moving objects are involved, then there is a need for temporal information. Take the example of driving in a queue in comparison to approaching a non-moving car at high speed. Using only spatial information as CNN does, the model can't distinguish those two scenarios from each other. The result should either be that the car is braking too much when driving in a queue, or that it is unable to break in time when it approaches a non-moving car with high speed.

3 Methodology

3.1 Overview

The methodology section will describe the setup used during the research. It will clarify the choices made and give a brief justification. An effort has been made to avoid theory in the methodology section, but the choices have been made using the theory outlined in the previous section. Some of the choices have also been justified by a try and fail approach, which has to do with the underlying nature of deep end-to-end learning. It can be thought of as a black box, where it is hard to find out why something worked well or didn't. These justifications will be further elaborated on during the discussion.

3.2 Research plan

The research was divided into two phases, exploration and exploitation. During exploration, the main goal was to solve three main task.

1. Finding a suitable feature extractor that would be complex enough to be able to distinguish relevant features in an image and simple enough to be able to process images fast enough to react within time during high speed.
2. Find three architectures (spatial, spatiotemporal and temporal) that has the following characteristics:
    - The architectures needs to handle multiple inputs and multiple outputs
    - The architecture should be able to provide a response within 0.25 seconds, which is the average reaction time for a human based on visual input. This limits the complexity of the architecture and the size of the input image.
3. Find a configuration that can learn the task of driving in the same town as it was trained in without any cars at an acceptable level.

Once this task was solved the next step was exploitation. The focus during this stage was to further develop the architectures to solve new and more complex tasks. The exploitation research was done in a cumulative way to achieve understandable result. Due to the complexity of deep neural network, it was important to start small then add more complexity once a task was achieved. The following plan was followed during exploitation and it was executed for each architecture:

1. Finish a complete track in the town used for training without other cars.

2. Finish a complete track in the town used for training with other cars.
3. Finish a complete track in a town not used for training without other cars.
4. Finish a complete track in a town not used for training with other cars.
5. Test architectures against each other on a wide range of scenarios.

3.3 Tools and equipment

3.3.1 System setup

A brief explanation of system setup, including GPU, OS(Ubuntu), Cuda, Anaconda, Keras, pandas, etc. For this project, a computer was set up with Ubuntu 18.04 with an Nvidia driver installed on a GeForce GTX 1080 Ti GPU. The computer was configured with two virtual environments, one using python 3.7 to run the server side code of the simulator, and one using python 2.7 for the client side code. Keras, Tensorflow, and Numpy were used for machine learning, while Pandas were used for data handling and data storing. Image handling was done using OpenCV.

Computer	
CPU	Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
GPU	GeForce GTX 1080 Ti
OS	Ubuntu 18.04
Driver	NVIDIA version 418.40.04
CUDA	Version 10.1
Environment	
Environment management system	Conda 4.5.12
Python distributions	Client side: 2.7.16 Server side: 3.7.1
Machine learning tools	
Tensorflow-gpu	Version 1.13.1
Keras	Version 2.2.4
Numpy	Version 1.15.1
Pandas	Version 0.24.2
Simulator	CarlaVersion 0.9.5 (+ manually adding some new commits to the global planner)

Table 1: System setup

3.3.2 Simulator

When deciding upon simulator there were two potential candidates. Namely Carla and AirSim. Both seemed quite similar in terms of functionality and features and it was hard to decide upon the right simulator to use for the project. Eventually, Carla was chosen based on two factors. In the discovery phase, more research was found were Carla was leveraged, which indicated more support in the research community. The other reason had to do with the support of operating systems. Carla supported mainly Linux while AirSim supported mainly Windows.

Given the fact that Carla is still in alpha stage, a lot of the project has been centered around writing software that collects training data and test models in a simulator environment.

3.4 Data gathering

3.4.1 Environment

Simulator configuration

The simulator environment was run without any display during data gathering to avoid resources being used on visualization. By doing this, the server could run above 100 fps when no other cars was spawned but usually lied in the range of 15 - 70 fps depending on the amount of actors in the simulation.

Weather

During training, a subset of weather was randomly chosen with a higher probability of getting cloudy. The reason to amplify episodes with good weather, was that important features as lane markings were not visible during weather like heavy rain.

Routes

During the gathering of data, a random start and stop position was fetched for each episode. The outer roads of the town were the only one that included speed limits above 30 km/h, and also the only ones that had turns without traffic lights. Unfortunately, these roads where rarely included in episodes, since the fastest way often was through the city. Thus, to get more data from the edge cases, routes that included the outer roads was chosen 20% of the time.

3.4.2 Agent

The autopilot

Data gathering was done using a modified version of the basic agent provided in the Python API. This is a client-side autopilot, which compared to the server side autopilot gives the developer more control, on the cost of worse driving. The agent performed poorly initially and several modifications were done to improve

the driving. Issues with the initial driving was:

1. The car would only brake if an obstacle was within 10 meters, which led to irregular driving. E.g heavy breaking then heavy acceleration, instead of trying to adapt the speed of the car in front.
2. If the car drove in a 90-zone and then encountered a 60- or 30-zone, it would only set throttle to 0, and not brake. This lead to several incidents where the car would have too high speed into a turn or crash into the rear of a car that followed the speed limit.
3. It had several problems with correctly identification of the correct traffic light in an intersection, which led to driving during red light and stopping halfway through an intersection because it thought the light on the other side was the one it should obey.

To solve these problems, some modifications were done to the controller to achieve training consistent training data without incidents or unwanted behaviour. Some of the major changes were:

1. Instead of only braking when a hazard was within 10 meters, the agent was modified to start braking from a distance of 20 meters if the vehicles speed was significant higher than the car in front. If the speed was not significant higher, then the target speed was set to the speed of the car in front, instead of the current speed limit. This improved the agent's ability to drive behind cars that drove below the speed limit.
2. The traffic light detection was rewritten, so that the agent did not get the unwanted behavior, with the tradeoff that it occasionally stops a little late.
3. Instead of only setting throttle to zero if speed was too high, it would now brake if the speed was 5 km/h above the speed limit.

The agent was also modified to run as fast as possible, thus the agent was stripped of all code that had anything to do with visualization. Then a recorder class was added to the agent to gather the training data.

Speed limit complication

All of the other cars in the simulation drives 10 km/h below the speed limit. This inferred complications for the autopilot. Should the autopilot and therefore also the trained model follow the actual speed limit or always drive 10 km/h below. Driving in 20 km/h vs 30 km/h or 80 km/h vs 90 km/h makes a huge difference in terms of making it easier to drive. But it also became more unrealistic, and therefore the choice was made to keep the original speed limits.

As long as the cars drove on a straight road, the autopilot would follow the speed of the car in front. But when the cars entered a turn, the autopilot didn't detect a car in front any longer, and therefore tried to speed up to 30 km/h. Halfway

through the turn it would either crash into the car in front or brake heavily. This inferred a degree of unwanted noise into the dataset. To prevent this the speed was lowered to 20 during driving in intersection.

Images

Data was collected using a RGB camera mounted on the front of the car. It recorded one image every 100 ms(10 fps) with the dimensions 345x460. It was placed right in front of the front window, thus it has a blind spot right in front of the bonnet.

Measurements

Together with the images a wide amount of measurements were stored. All of the measurements are displayed in table 2. These measurements were either used for debugging, input to trained model, or as targets during training.

Noise injection

Noise was injected to the data during driving to get mores samples where the car is correcting itself. Noise was added using momentum and step. It was randomly injected during driving, and once applied the noise would be produced by the following equation 0.025*step. This was applied 8 times for each noise injection. Forcing the car to drive right or left for a short period, then correction is done by the autopilot.

Measurements recorded	
Frame	The corresponding image frame.
Speed	The current speed of the car in km/h
Throttle	Floating variable between 0 and 1, which corresponds to the acceleration of the car.
Steer	Floating variable between -1 and 1 which tells the amount of steering to the left or right
Brake	If the car is braking or not represented as 1 or 0.
Gear	Gear of the car
Speed_limit	The current speed limit
At_TL	Boolean that states if there is a traffic light within 10 meters.
TL_state	The current state of an upcoming traffic light, always green unless red traffic light within 10 meters.
FPS	Average server-fps for the last 100 frames
Direction	Which direction the car should take in an intersection
Real-time(s)	Actual time since initialisation of the agent
Simulator-time(s)	Simulator time is different from real time in the way that the server might slow it down to achieve a wanted frame rate. This happens when benchmarking is applied to the server.
Controller updates	Amount of controller updates sent to the server

Table 2: recorded measurements

3.5 Data preprocessing

3.5.1 Data preparation

The data gathered from driving could not be used directly and some steps was done to prepare it before training. Images gathered during training was first cropped by removing the top 165 pixels. This was to remove the part of the picture that contained unnecessary information for the task. After cropping the images was resized to 66x200 pixels. For the measurements a one-hot-encoding was applied to data that could be divided into categories as for example Traffic light state, direction and speed limit. While continuous variables as speed were normalized.

The direction command for intersections was prepended up to 30 timesteps backwards to prevent the car from learning only the relationship between direction and steering. Instead it has to understand using features from the image that it is in an intersection.

3.5.2 Filtering

One of the most important prepossessing steps was to filter out a large amount of the data. This has to do with the environment the car is driving in. Most of the driving is straight forward driving with no intersection. Filtering was applied to remove samples where the car stood still (waiting due to red traffic light), and where steering was approximately 0. Different filtering sizes was applied, but usually a threshold around 0.7-0.9 was applied.

3.5.3 Data augmentation

Filtering away to much of the samples where steering is 0 might result in loss of important information used to fit throttle and brake. Data augmentation was therefor applied to create a more balanced dataset without removing samples. The following samples were augmented:

- Absolute steering higher than 0.5 were multiplied 5 times
- Absolute steering higher than 0.1 were multiplied 2 times
- Samples where braking when traffic light is green were multiplied 3 times
- speed limit 60 samples were multiplied 3 times and speed limit 90 samples were multiplied 2 times

3.5.4 Training data statistics

As can be seen in figure 3 and 4, the unfiltered data is not as balanced as one would like. This is very evident, especially from the steering distribution. Therefor, the data was filtered and augmented producing the distributions seen in figure 5 and 6.

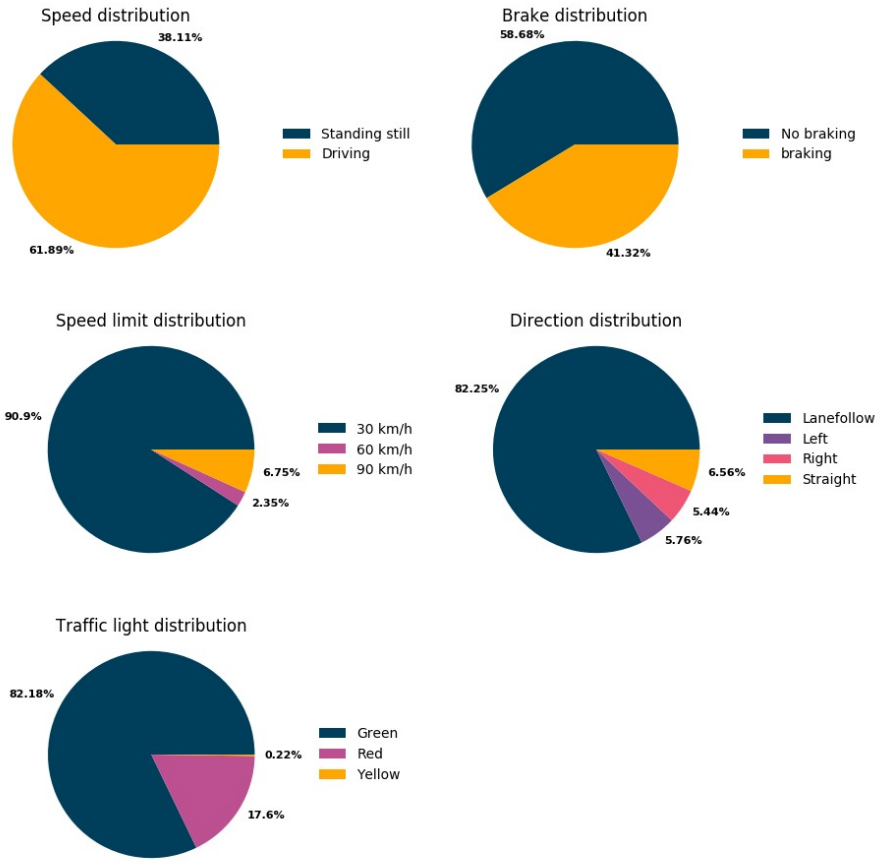


Figure 3: The following pie charts describe the characteristics of the training set before filtering and data augmentation is applied.

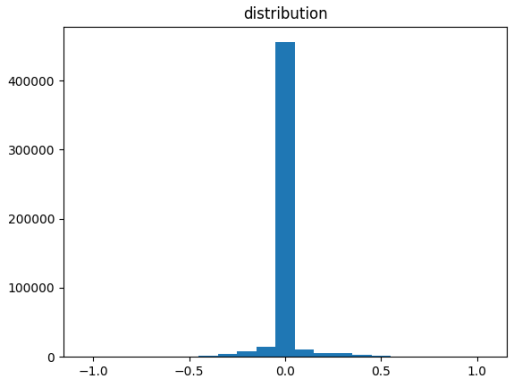


Figure 4: Histogram showing unfiltered steering distribution in training set

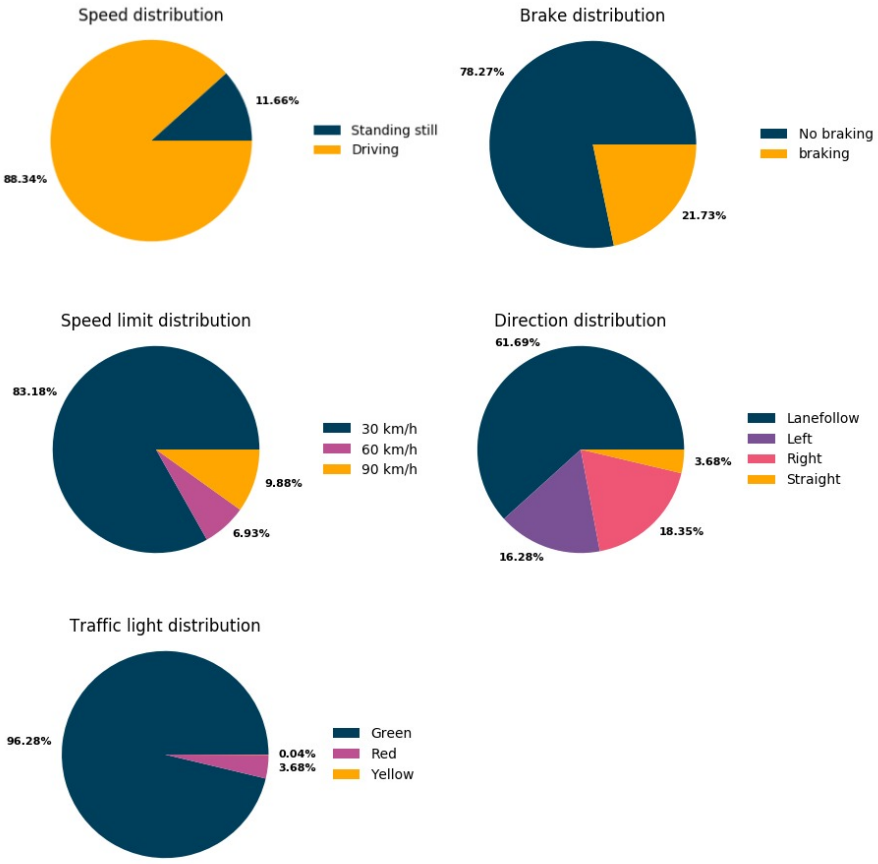


Figure 5: Pie-chart describing characteristics of training set after filtering and augmentation

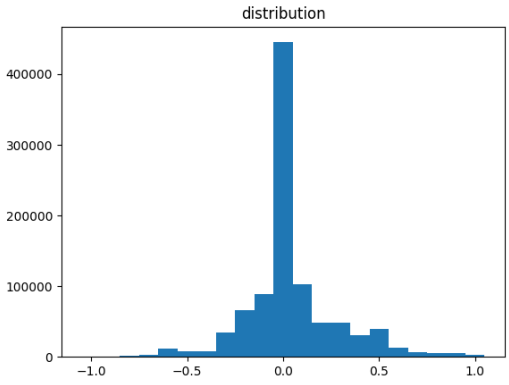


Figure 6: Histogram showing steering distribution in training set after filtering and data augmentation

3.6 Training

To achieve efficient training on a large amount of architectures, several steps was taken before training and testing were started. During exploration, a grid search was applied to test a lot of combinations. This was to find a set of hyper-parameters and network architectures that could be used as basis during exploitation.

3.6.1 Network architectures

The architectures applied for the task all have in common that they have multiple inputs and multiple outputs. They use CNN to extract features from images, and then merge these with the rest of the inputs. On the other side they output values for steering, brake and throttle.

During training, three different architectures were applied. The first one, spatial 7, is a CNN taking the newest image as input, feed forwards it through a CNN, concatenates it with the measurements, then forwards the features to a dense top. The second, named spatiotemporal 8, is an architecture that combines multiple timesteps by stacking several of the spatial architectures. The third architecture, named Temporal 9, is a RNN that takes several timesteps as inputs, which also uses the same ground structure as used in spatial, but the CNN is now timedistributed and not stacked.

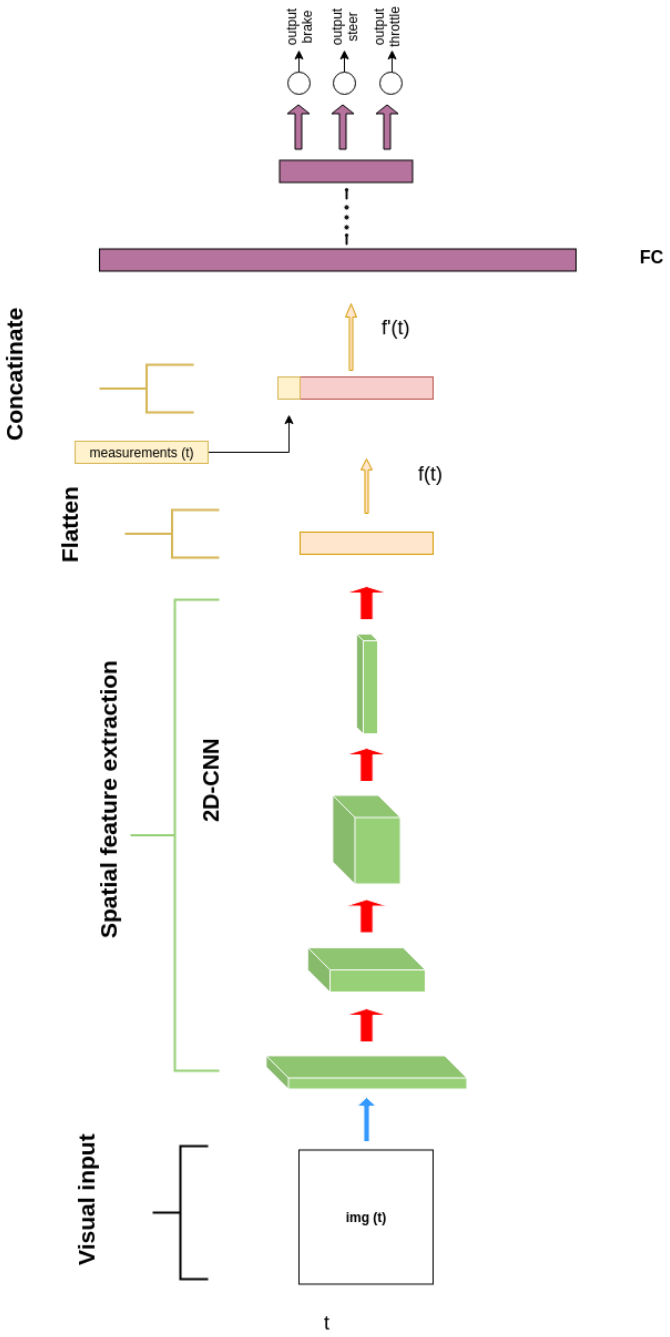


Figure 7: The architecture used for Spatial training

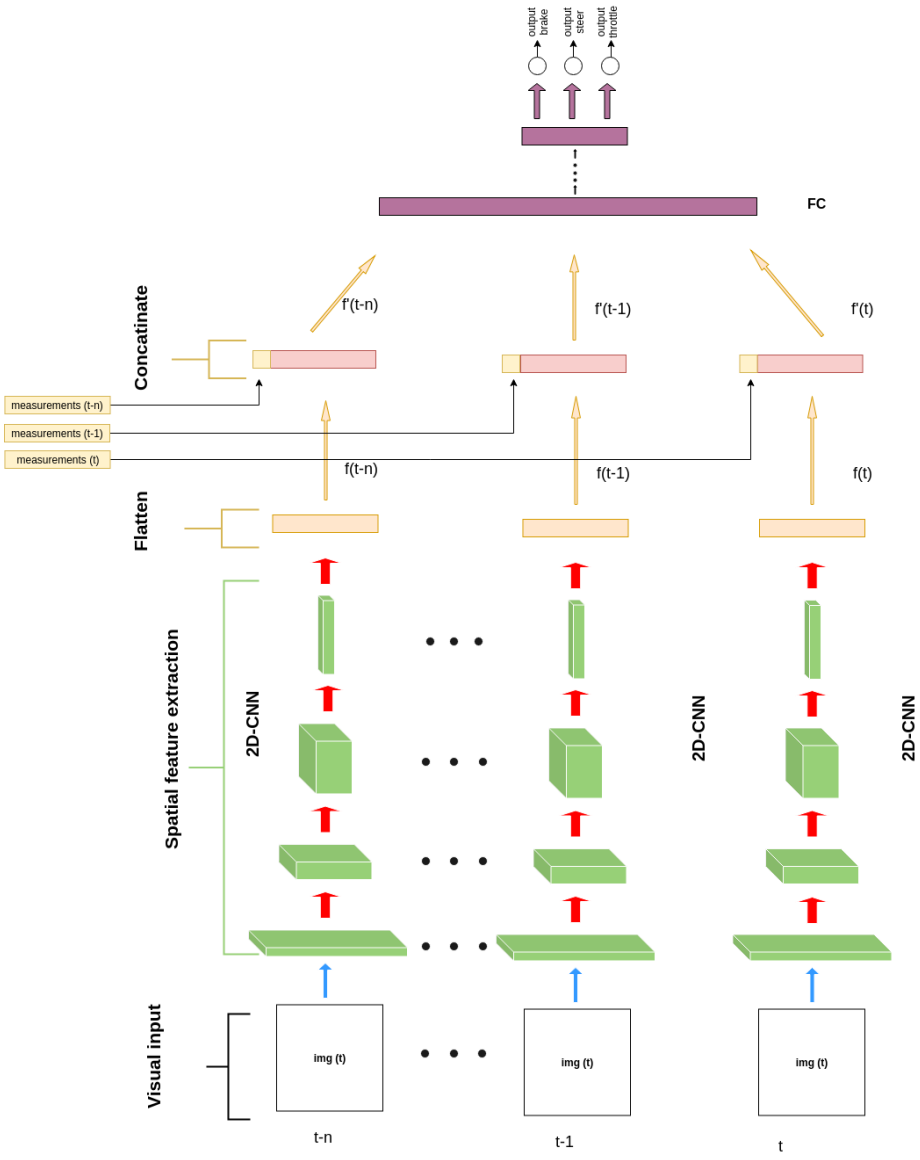


Figure 8: The architecture used for spatiotemporal training. Each CNN contains its own weights.

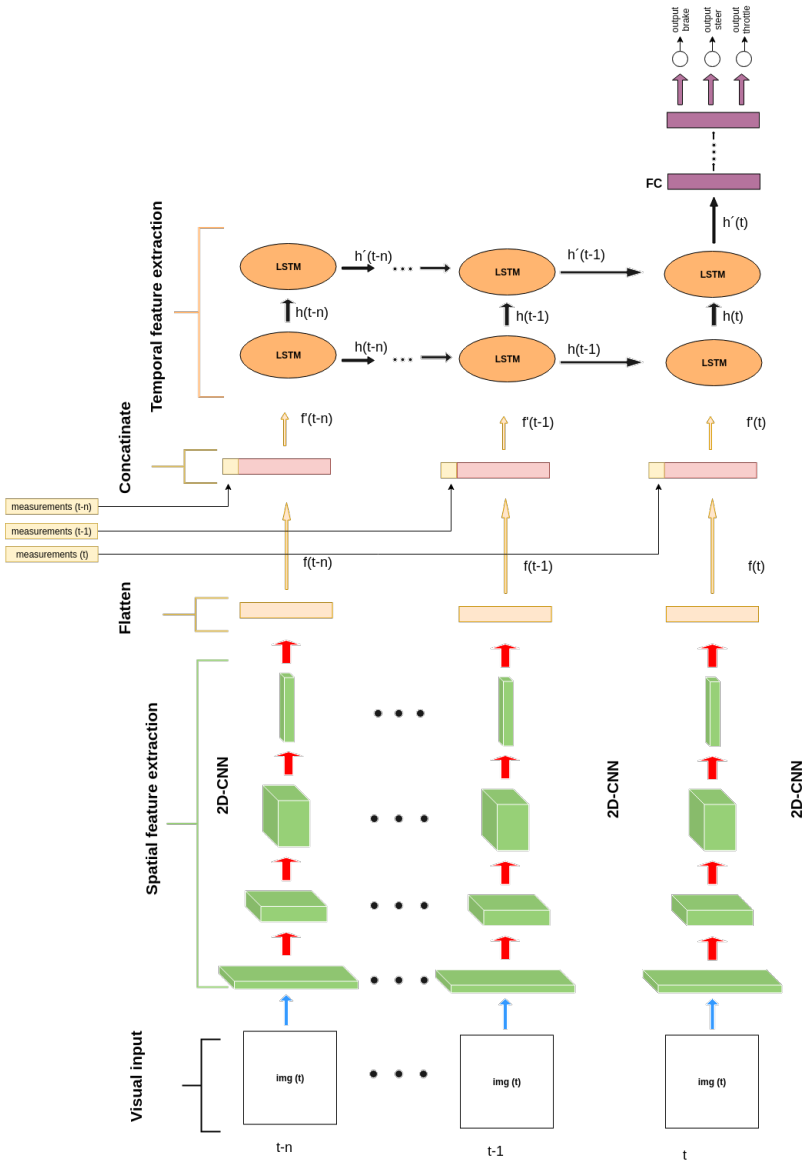


Figure 9: The architecture used for temporal training. Note that the CNNs' are actually wrapped in a time-distributed layer, which means that the same weights is used for each timestep, and there is not stacked CNNs' like in Spatiotemporal

3.6.2 Callbacks

During training, every epoch that achieved better validation results than previously achieved, was stored as a checkpoint. To prevent overfitting two callbacks

were implemented to stop training early. One that lets the user stop the training manually if needed, and one that automatically stops training if validation error increases. At the end of the training all necessary information to reproduce the results are stored combined with a model plot, the checkpoints and a loss plot.

3.6.3 Loss functions

Binary cross entropy and MSE was tested as loss functions during training. Both were applicable since braking was applied by the autopilot as either on or off. Steering and throttle are in comparison continuous variables, thus only MSE was applied on these outputs.

3.6.4 Training and validation set

The training set consisted of 512 000 samples, while the validation set contained 160 000 samples. During one epoch only a subset of the full dataset was provided. The training size was for example set to 4000, resulting in 64 000 samples randomly sampled during each epoch given a batch size of 16. The training setup was very dependent on a validation loss that represented the full validation set. A size of minimum half the training size were therefor used, resulting in 32 000 randomly sampled samples each epoch.

3.6.5 Optimizer

RMS-propagation was used heavily combined with hyperbolic tangent(tanh) - as activation function for the convolutional and the dense layers - during exploration of the temporal architecture. While Adam combined with rectified linear unit (relu) as activation function, was mostly used for the spatial and spatiotemporal architecture.

3.6.6 Grid search

During training, a grid search to find the optimal hyperparameters was applied. The search tested different variations of networks, epochs, batch-sizes, learning rates, sequence lengths and filtering degrees.

3.7 Testing

3.7.1 Setup

The final testing of the architectures used a range of 8 weather conditions and 28 tracks and tested the combination of all of them, with and without cars. This totalled to an amount of 448 test runs, covering everything from only straight driving, to driving with a combination of high speed and inner city maneuvering. Figure 10 shows a bird view of town 2 filled with the available spawn points and should be seen in combination with table 3 which shows the spawn points used

during testing.

Figure 10: Figure showing the available spawn points in town 2

Track	Description	[Start, Stop]
0	low speed straight	[49, 45]
1	high speed straight	[18, 25]
2	intersection left	[57, 76]
3	intersection right	[46, 72]
4	turn and intersection right	[40, 62]
5	intersection and turn left	[63, 39]
6	circle inner and outer right turns	[43, 68]
7	circle inner and outer left turns	[35, 69]
8	right to left turn	[44, 57]
9	left to right turn	[66, 46]
10	high speed with turns	[64, 11]
11	high speed with turns opposite	[9, 75]
12	outer track 1	[64, 36]
13	outer track 1 opposite way	[37, 75]
14	outer track 2	[38, 11]
15	outer track 2 opposite way	[9, 39]
16	Through the city 1 long	[82, 5]
17	Through the city 2 long opposite	[7, 32]
18	inner city short 1	[70, 56]
19	inner city short 1 opposite	[66, 71]
20	inner city short 2	[44, 76]
21	inner city short 2 opposite	[72, 45]
22	outer to inner city 1 difficult	[75, 71]
23	outer to inner city 1 difficult opposite	[9, 45]
24	outer to inner city 2 difficult	[37, 63]
25	Through the city 2	[37, 81]
26	High speed then through the city	[12, 6]
27	inner city easy	[46, 45]

Table 3: The table shows the tracks used for final testing and should be seen in combination with figure 10

3.7.2 Metrics

If the track was completed or not isn't by itself a sufficient measure of good driving and therefor a set of metrics were defined for testing purpose. The metrics used, found in table 4, gave a better understanding of the actual driving performance of a model.

Metrics	
Steer-score	The absolute mean of all the turning signals throughout the episode. The more unsteady the model is, the higher the score.
Speed-score	Mean STD of the speed relative to the speed limit.
Speed-under-score	When speed was lower than the speed limit, what was the mean deviation.
Speed-over-score	When speed was higher than the speed limit, what was the mean deviation
Crossed-line-score	Number of times the car crossed a line.
Collision	If the car collided during the episode. (Collision leads to termination of the episode and is also used as a measure to know if the track was completed or not)

Table 4: Metrics for testing

3.8 Configuration used for final testing

The following section gives a brief overview of the most important configurations that were used to achieve the final models that were compared. As can be seen in figure 5, 6 and 7, a lot of the same configurations ended up working best for all architectures. The configurations that had most to say, were the top dense layers for all, and the sequence length and steps size for the temporal.

Spatial	
Parameter	Value
Learning rate	5.2 e-05
Optimizer	Adam
Loss	MSE
Loss weights steer	4.0
Batch size	16
Steps per epoch	4000
Validation steps per epoch	2000
Filtering degree steering	0.8
Top layers	512, 256, 16, 1

Table 5: Final configuration of the spatial model

A weighted loss function, was first inferred during exploitation, based on the fact that the steer loss had one order of magnitude less than the output loss and

Spatiotemporal	
Parameter	Value
Learning rate	5.2 e-05
Optimizer	Adam
Loss	MSE
Loss weights steer	4.0
Batch size	16
Steps per epoch	4000
Validation steps per epoch	2000
Filtering degree steering	0.8
Sequence length	4
Step size	2
Top layers	512, 256, 64, 16

Table 6: Final spatiotemporal training configuration

the throttle loss.

Temporal	
Parameter	Value
Learning rate	5.2 e-05
Optimizer	Adam
Loss	MSE
Loss weights steer	4.0
Batch size	16
Steps per epoch	4000
Validation steps per epoch	4000
Filtering degree steering	0.8
Sequence length	10
Step size	2
Activation function	relu (tanh for LSTM)
Top layers 256, 128, 16	

Table 7: Final temporal training configuration

4 Results

This chapter will be divided into two parts. Training results and test results. Section 4.1 is included to give a brief overview of the results from the training of the final models and architectures, while section 4.2 will provide results from the final testing done to compare the architectures.

4.1 Training results

The final models used for testing all achieved a validation loss below 0.1. After this point the plotted losses are starting to show signs of overfitting, which can be seen in figure 11, 12 and 13, where the validation loss starts to flatten out while the loss continues to decrease.

The spatial model has the most consistent results from training and achieved a validation loss of 0.065 after 28 epochs. There is little fluctuation, and the training showed a gradually decrease in loss with time. As can be seen in figure 11, it gradually starts to show signs of overfitting after epoch 10.

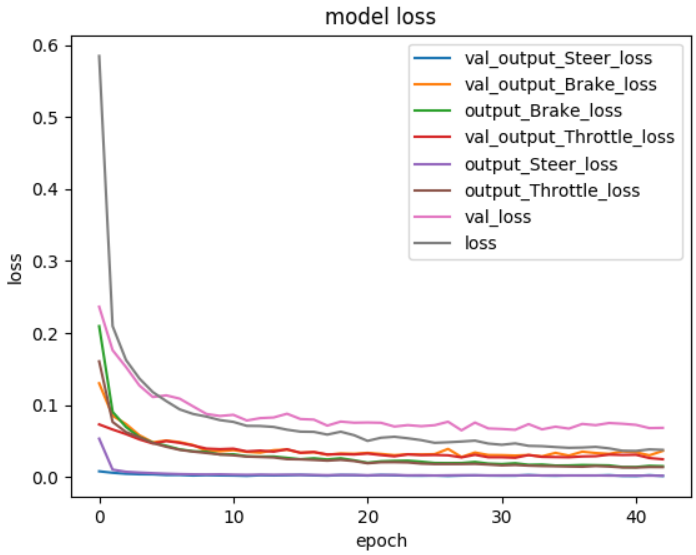


Figure 11: Graph showing training and validation loss from training of the spatial model

The spatiotemporal model had more fluctuation in terms of validation loss, but manages to achieve a better loss before showing signs of overfitting. The best checkpoint in terms of validation loss is at epoch 37 with a validation loss of 0.045.

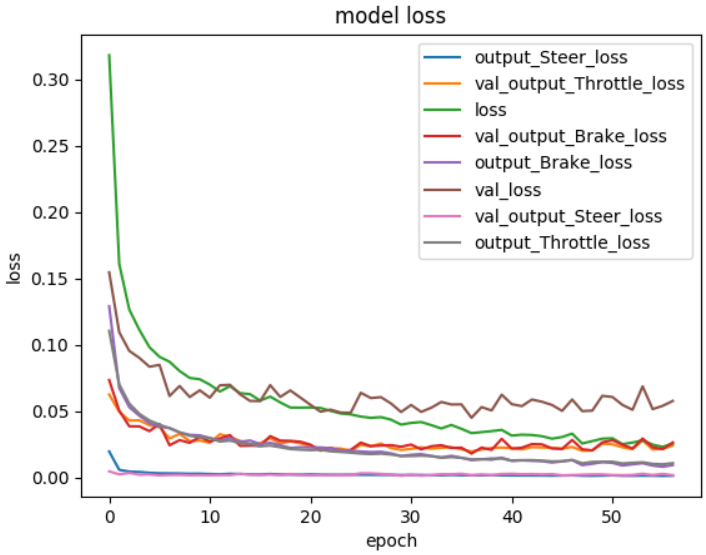


Figure 12: Graph showing training and validation loss from training of the spatiotemporal model

The temporal model uses the most time to converge towards the target function, achieves a loss of 0.1 after approximately 25 epochs, for comparison, the spatiotemporal model achieved a validation loss below 0.1 after 3 epochs. The best results achieved in terms of validation loss was after 30 epochs, with a validation loss of 0.087.

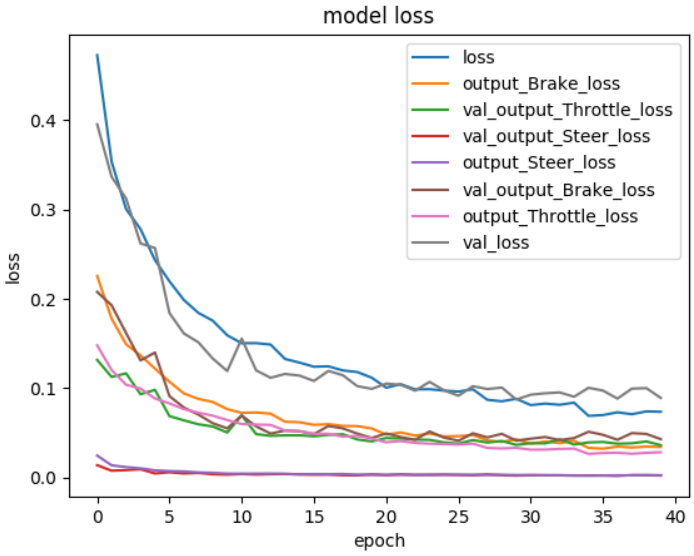


Figure 13: Graph showing training and validation loss from training of the temporal model

4.2 Testing results

Testing showed that the spatiotemporal model achieved best results overall, which can be seen in table 8. It completed more than 50% of the episodes used for testing and had the lowest speed score. The spatial model completed 38% of the episodes, but had the highest count of crossed lines.

	Spatial	**Spatiotemporal**	**Temporal**
Test episodes	448	448	448
Targets reached	169	263	130
Completion degree	37.70%	58.70%	29.00%
Crossed line count	4.15	4.07	3.89
Mean distance driven	167.05	191.34	163.62
Speed score	3.43	3.35	6.46
Steer score	5.211	4.98	3.97
Model	9	13	25
Epoch	23	16	26
Loss	0.07	0.06	0.09

Table 8: Overall statistics from the testing based on network architecture

The spatiotemporal model struggles a lot with other cars and completed less than half of the episodes that it completed without cars. Still, it outperforms both the spatial and spatiotemporal model with and without other cars.

No cars			Cars		
Spatial	**Spatiotemporal**	**Temporal**	**Spatial**	**Spatiotemporal**	**Temporal**
107	178	70	62	85	60

Table 9: Targets reached based on driving with or without other cars

All the models drive quite consistently in all weather types except for heavy rain, which has less than half the completion rate compared to the other weather types.

	Spatial	**Spatiotemporal**	**Temporal**
Clear noon	23	43	12
Cloudy noon	29	44	21
Wet noon	26	34	16
Soft rain noon	20	35	21
Wet cloudy noon	20	27	16
Mid rainy noon	22	34	19
Hard rainy noon	9	7	6
Clear sunset	20	39	19

Table 10: Targets reached based on weather conditions and network architectures.

Overall there was a gradually decrease in episode completion when the difficulty of the tracks was increased. While the first tracks were achieved under almost every condition, only a small subset of the episodes ended in completion on the last tracks.

Network architecture				
Track	**Track description**	**Spatial**	**Spatiotemporal**	**Temporal**
0	low speed straight	16	15	15
1	high speed straight	14	15	15
2	intersection left	1	15	4
3	intersection right	12	15	16
4	turn and intersection right	14	14	14
5	intersection and turn left	14	14	13
6	circle inner and outer right turns	0	5	0
7	circle inner and outer left turns	1	11	3
8	right to left turn	10	8	8
9	left to right turn	8	14	4
10	high speed with turns	9	13	1
11	high speed with turns opposite	4	9	3
12	outer track 1	6	9	0
13	outer track 1 opposite way	0	8	1
14	outer track 2	8	8	0
15	outer track 2 opposite way	3	7	2
16	Through the city 1 long	7	8	0
17	Through the city 2 long opposite	5	10	5
18	inner city short 1	2	10	2
19	inner city short 1 opposite	8	8	6
20	inner city short 2	5	8	2
21	inner city short 2 opposite	9	7	6
22	outer to inner city 1 difficult	0	4	0
23	outer to inner city 1 difficult opposite	0	4	0
24	outer to inner city 2 difficult	1	6	1
25	Through the city 2	1	5	2
26	High speed then through the city	0	5	0
27	inner city easy	11	8	7

Table 11: Targets reached based on track and network architecture.

5 Discussion

5.1 Feature extractor and input size

During exploration, several different variations of feature extractors were tested, but best results were achieved using the same architecture as defined by Nvidia [7]. Attempts were done to use a higher resolution image and a more complex feature extractor. This led to acceptable results with the spatial and spatiotemporal architecture, but it was very difficult to find a configuration that enabled the temporal model to learn.

During exploration it was found that an increase in image size was way more critical to the training than anticipated. Increasing the image size from 66x200 pixels to 130x320 pixels would increase the input size to the feature extractor from 13.200 to 41.600. But after the same feature extractor (in this case the Nvidia architecture) is applied to the image, then the amount of features extracted are increased from 1.152 features to 19.008 features. If you then add the fact that you are working with a spatiotemporal model with for example a sequence length of 5, then you will find that the amount of features provided to the top layers have increased from 5.760 features to 95.040 features. This increase in features also results in an increase to the models complexity. During exploration a typical output size on the layer after the feature extractor was 512. Thus, we had an increase in the amount of parameters to train from approximately 1.5 millions to approximately 49 millions. Its hard to discuss the actual effect of such an increase in parameters, but as can be seen in image 14, there is not a lot of difference in terms of information available. And if it is possible to fit the target function using 1.152 extracted features, then a guess might be that the increase of features might infer a large amount of noise.

Figure 14: The same input image is here displayed at the top using 130x200 pixels and at the bottom using 66x200

5.2 Dataset used for training

5.2.1 Validation set to similar to training set

As mentioned in section 4.1, the best validation loss was achieved at epoch 28, 37, 30, but when the models were tested in town two, the best results were achieved at epochs 23, 16, 26. This can be seen in figure 20 under appendix B. These results makes it clear that the validation set isn't sufficiently different enough from the training set to capture overfitting at an early stage. To better capture the occurrence of overfitting, one should include a third town, slightly different from the two others, and use this for validation. This would most likely have improved the early stopping of the model, and instead of testing several epochs too see which was better, one could would a high degree of certainty use the epoch with best validation loss.

5.2.2 Achieving consistent loss convergence with large amount of training data

A large training set, consisting of around half a million training samples were gathered to allow for a lot of variance in the data provided during training. A result of this was that training on the full dataset became more or less unfeasible with the initial configuration. The problem encountered was that the amount of parameter-updates during an epoch was so large that it was difficult to stop the model at the point were it would begin to overfit.

To accommodate such a large dataset, a normal approach is to increase the batch size. Doing this actually led to worse performance during training, and the model would struggle hard to reach the same loss values as with a lower batch size. Therefor a few modifications to the training configuration was applied. During one epoch of training a model will usually be exposed to the full dataset available. This was changed such that a random sample of 4000 batches were used instead. This approximates to 8% of the full dataset. Using such a small sub-sample led to a lot of fluctuation between the epochs based on the characteristics of the randomly sampled batches. To prevent this, the learning rate was decreased to $5.2 * 10^{-5}$, to allow for more stable convergence. The result was a loss that converged at a steady rate without too much fluctuation, as can be seen in figure 11, 13, 12.

5.2.3 Garbage In Garbage Out

Garbage-In-Garbage-Out in this context means that the trained model can never be better than the autopilot it learned from, and the autopilot used for collection of training data is not optimal. During data gathering it crashed into the car in front in one out of three episodes. During testing, several of the incidents happens in the same scenarios as the autopilot had troubles with during data gathering. Figure 16 illustrates this issue well. In this scenario the autopilot would quite often crash. This is because it cant detect any cars in front, because they are on a road with a different road ID (The autopilot uses road ID to detect cars in front). Similar, when the car is driving in high speed (60-90km/h), then it struggles to stop in time for slow moving vehicles or cyclists. This can be seen in figure 15.

5.2.4 Speed limit

During exploration it was tested to drive with the same speed limit as the other cars(10km/h below the actual speed limit). This made the task of driving easier, and the target function that the model should learn was less complex and easier to fit. Anyways, the actual speed limit was chosen to use for training and testing and the reasoning behind this was that it inferred some wanted temporal dependencies. When driving with the modified speed limit (10km/h below the original), the autopilot and the trained model would rarely end up driving behind a car. It

would meet cars standing still in front of an intersection, but rarely elsewhere. When driving with the original speed limit, the autopilot and the agent would often catch up to the car in front while it where driving, and thus have to lower its speed to the same speed as the car in front then keep that speed. Following the speed of a car in front is something that is quite hard to do using only spatial information as mentioned during the sub-section Spatial vs Temporal information. The spatial model would brake if it thought it was to close to the car, and gas once it was far enough away, thus creating sort of a yo-yo effect during driving. The spatiotemporal model obtained the speed of the car in front easier, and were able to maintain it without excessive braking.

The result is that the driving behavior became quite aggressive related to the other cars. The testing became more difficult, as for example turning at a higher speed or braking fast enough when approaching a turn in high speed. The target function became more complex in terms of throttle and braking. But the outcome was more realistic testing scenarios to use for comparison between spatial and temporal models.

Figure 15: The image shows two time-steps from an episode recorded during data gathering. As can be seen, the autopilot isn't able to detect the cyclist early enough and therefor fails to stop in time.

Figure 16: The image shows 9 time-steps from an episode recorded during testing, where the car ends up crashing into the car in front

5.3 Comparison of architectures

This section will take a look at the architectures tested and make a comparison based on their characteristics. The main focus will be between the Spatiotemporal and temporal model, since these are of most interest in this thesis.

5.3.1 Test result

The best architecture was without doubt the spatiotemporal, outperforming the other architectures in almost every scenario and metric. That being said

5.3.2 Complexity of fitting a good model

The spatial model was in comparison to the others easy to fit and to get to achieve acceptable results. Once it drove good without cars and acceptable with cars, then most of the focus was put on the temporal and spatiotemporal models. These were harder to find a good architecture for, mostly because the increase in complexity. A necessity for the spatiotemporal model was to find the correct size of the dense

layers. Except for that the architecture is similar to the spatial one, except for the stacking of feature extractors. After a good combination of dense layers was found, then tweaking of hyper-parameters was applied to get the final model. This was not the case with the temporal architecture. This architecture was without doubt the hardest to train, and to get to achieve a validation loss below 0.1. This doesn't mean that the temporal architecture is less suited for the task, but the complexity of a recurrent neural network makes it more difficult to configure. If the correct architecture and training configuration was found during exploitation, then a certain guess would be that it would achieve better results on the task compared to the Spatiotemporal model. This comes from the fact that an LSTM uses a forget gate that enables it to filter what kind of features it should pass forward from earlier time-steps to the dense layers, while the spatiotemporal model provides all the feature from all the time steps to the dense layers.

5.3.3 Model complexity

The forget gate applied in the LSTM makes a huge difference in terms of model complexity. The final spatiotemporal model had 3 million parameters that needed to be trained while the temporal had 1.5 millions. The difference is not to large, but the temporal models' complexity is independent of the sequence size. So lets say that we would like to keep information from the past 5 seconds, meaning a sequence length of 25, then the Temporal model would still have a 1.5 million parameters, but the spatiotemporal would have at least 15 million, not taking into consideration that the dense layers would have to be increased to accommodate the new size of features extracted.

5.3.4 Prediction time

As fast reaction time is important when driving, the time used for prediction is also quite important when comparing models. Under section 3.2, one of the wanted characteristics was that the reaction time on visual input would be better than that of a human. All of the models actually achieved way beyond human reaction time, and they were actually limited by the framerate of the simulator. All of the models therefor made 20 predictions per real time second, resulting in a reaction time of 5ms. That being said, a comparison of what would happen if we increase sequence length and or image size is in good nature.

To produce the output from one LSTM cell (at time step t), one would have to first get the output from the previous LSTM cell (at time step t-1). Thus each timestep needs to be processed sequentially, inferring an increased time in prediction speed when the sequence length is extended. The LSTM layer used for this project is not optimized for GPU either, but TensorFlow have developed a CudnnIstm cell that is backed by Cudnn and therefor supports GPU. This layer is currently limited in

terms of available activation functions and perks like dropout, but at the other side produces a great amount of increase in training and prediction time.

The Spatiotemporal model can on the other hand fully use the advantages that follows with GPU support, and if enough hardware is available, then it should be possible to parallel process all the feature extractions, resulting in minimal increase in prediction time from increased size.

The task of end-to-end driving would probably not need to leverage more than at max 5-10 seconds of previous information, and therefore it doesn't seem like there is any problems related to prediction time with the current architectures. A more likely barrier will probably be the increased complexity from leveraging higher resolution images and more complex feature extractors.

6 Conclusion

This thesis has first of all shown that a stacked Convolutional layer, that uses a timeseries as input can learn the task of driving in urban areas quite sufficient. The thesis shows that it is able to generalize well to new unseen environments, which confirms good generalization.

Further the thesis might indicate that the task of fitting a stacked CNN is less troublesome than the task of fitting a RNN, and it also produces noticeably better results compared to a Spatial model based on the same feature extractor. Thus, a stacked CNN is a viable alternative to RNNs, for the task of autonomous driving.

Since I couldn't achieve good enough convergence towards the target function with the RNN, I can neither confirm or refute which is better for the task of autonomous driving. But the thesis indicates that when the same feature extractor and configuration are used in the three models, then the spatiotemporal architecture achieves remarkable better results compared to the spatial and the temporal architecture.

6.1 Future Work

While the results are promising, there are several steps to take from here. First of all, to get closer to a complete end-to-end approach, it should be able to use visual input to figure out some of the measurements that where hard coded during this experiment. In particular, a camera pointed against road signs and traffic lights could be added to the car, and thus provide the architecture with enough visual input to be able by itself to figure out the speed limit and the current traffic light condition.

Seeing that one of the issues with a stacked CNN compared to a RNN is that it does not use a forget gate, it would be relevant to test the two models against each other on a driving task or a scenario where important information gets provided once. And the network needs to remember the information until it should act upon it. This can also be tested by providing the direction only once a given timesteps before an intersection to see if this is enough information to force the vehicle to follow the planned route.

An effort was done to increase the complexity and image resolution, to provide a better model. This was harder to fit and limited time on the thesis resulted in a final choice being a less complex model and a lower resolution image. Still, there is a lot of potential in using a more complex model which should be investigated.

Bibliography

[1] Codevilla, F., Muller, M., Lopez, A., Vladlen, K., & Dosovitskiy, A. 2018. End-to-end driving via conditional imitation learning. https://arxiv.org/pdf/1710.02410.pdf. Accessed: 2019-03-03.

[2] Pomerleau, D. A. 1989. Alvinn: An autonomous land vehicle in a neural network. https://papers.nips.cc/paper/95-alvinn-an-autonomous-land-vehicle-in-a-neural-network.pdf. Accessed: 2019-03-16.

[3] Chen, C., Seff, A., Kornhauser, A., & Xiao, J. 2015. Deepdriving: Learning affordance for direct perception in autonomous driving. https://arxiv.org/pdf/1505.00256.pdf. Accessed: 2019-03-20.

[4] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., & Koltun, V. 2017. Carla: An open urban driving simulator. https://arxiv.org/pdf/1711.03938.pdf. Accessed: 2019-03-11.

[5] The open racing car simulator. http://torcs.sourceforge.net. Accessed: 2019-03-23.

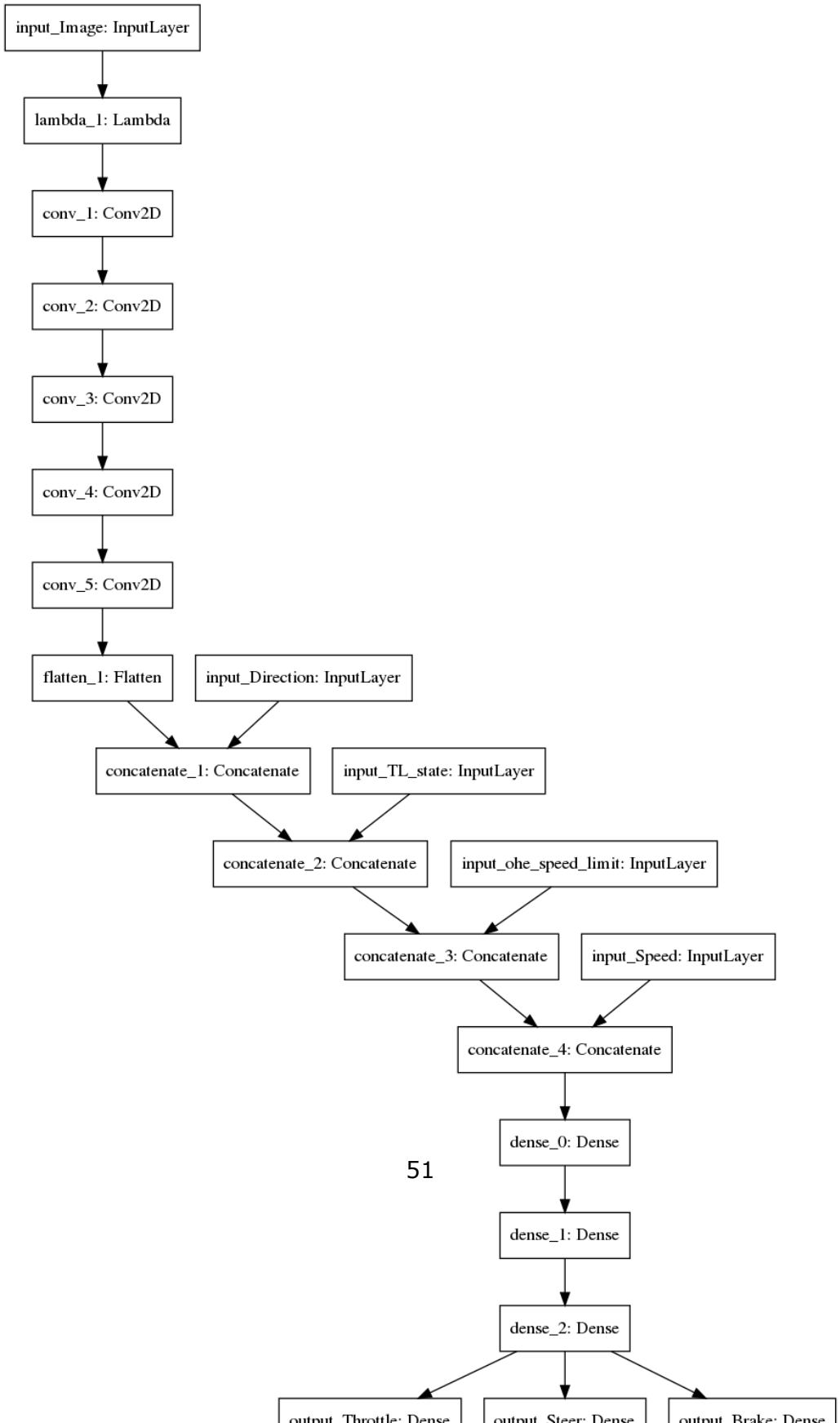
[6] LeCun, Y., Muller, U., Ben, J., Cosatto, E., & Flepp, B. 2005. Off-road obstacle avoidance through end-to-end learning. http://yann.lecun.com/exdb/publis/pdf/lecun-dave-05.pdf. Accessed: 2019-03-25.

[7] Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., & Zieba, K. 2016. End to end learning for self-driving cars. https://arxiv.org/pdf/1604.07316.pdf. Accessed: 2019-03-20.

A Exploitation results

A.1 Spatial

A.1.1 Model 9



A.2 Spatiotemporal

A.2.1 Model 13

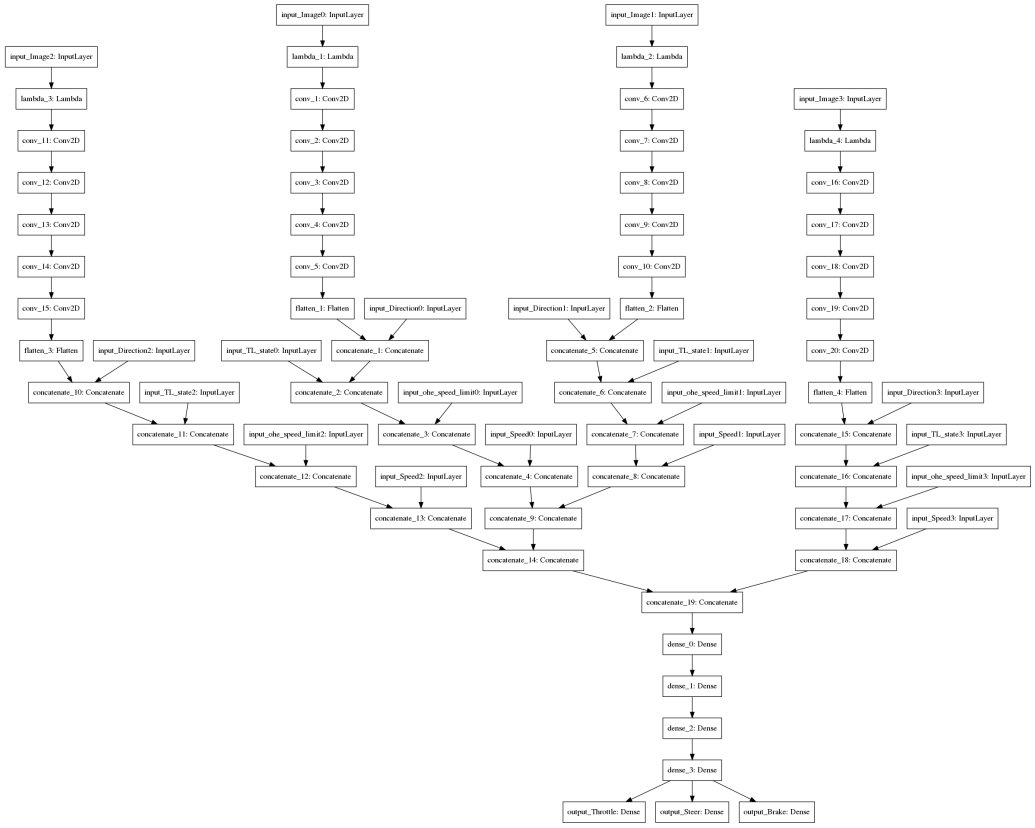


Figure 18: Final Spatiotemporal model architecture

A.3 Temporal

A.3.1 Model 25

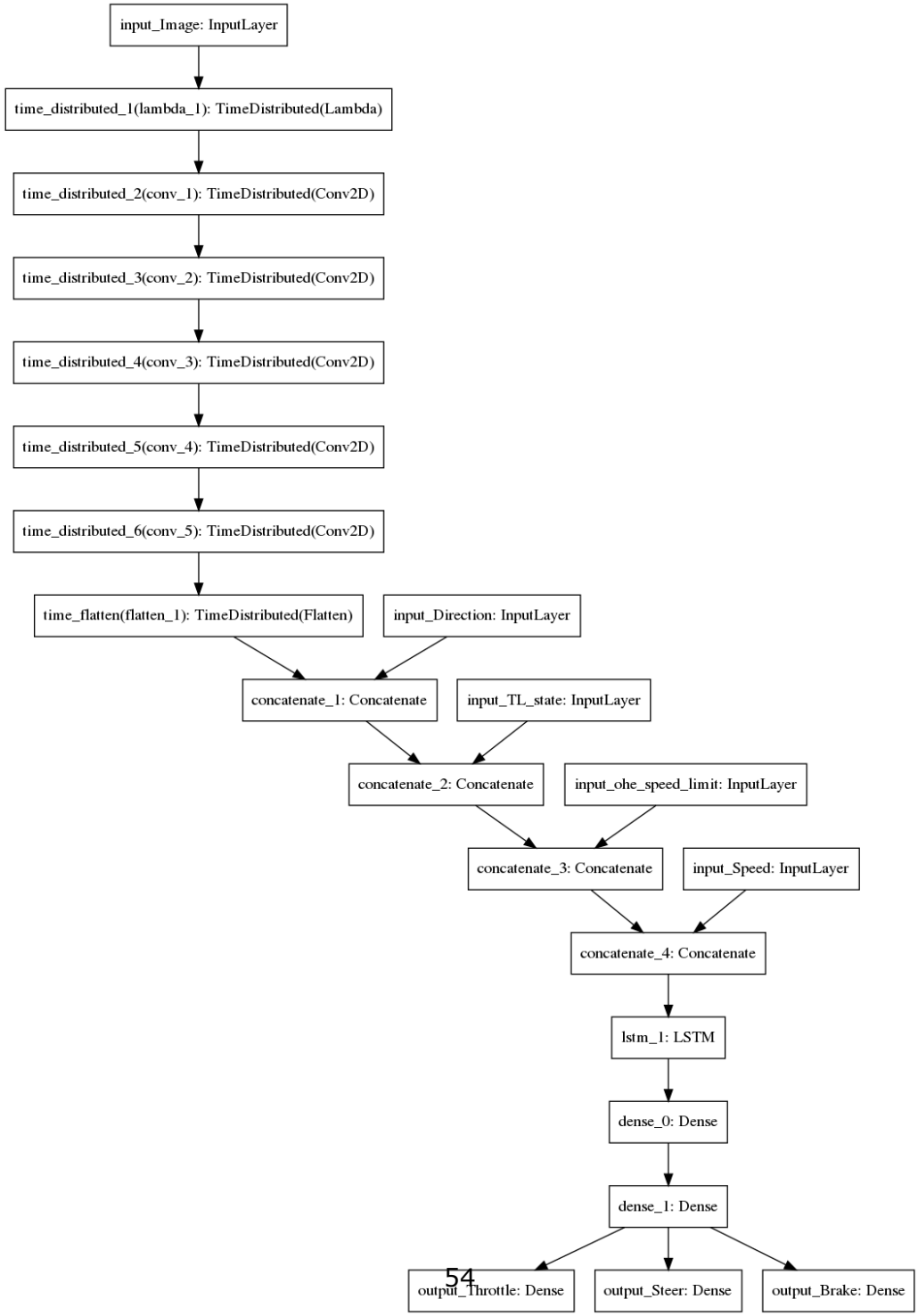
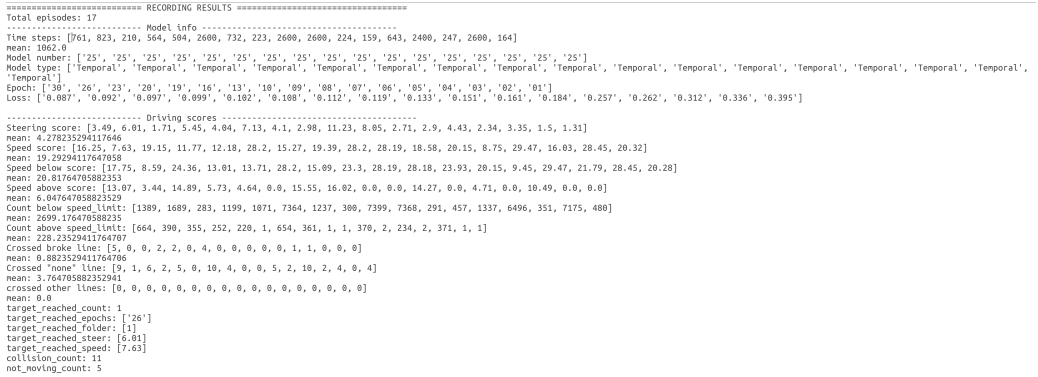


Figure 19: Final temporal model architecture

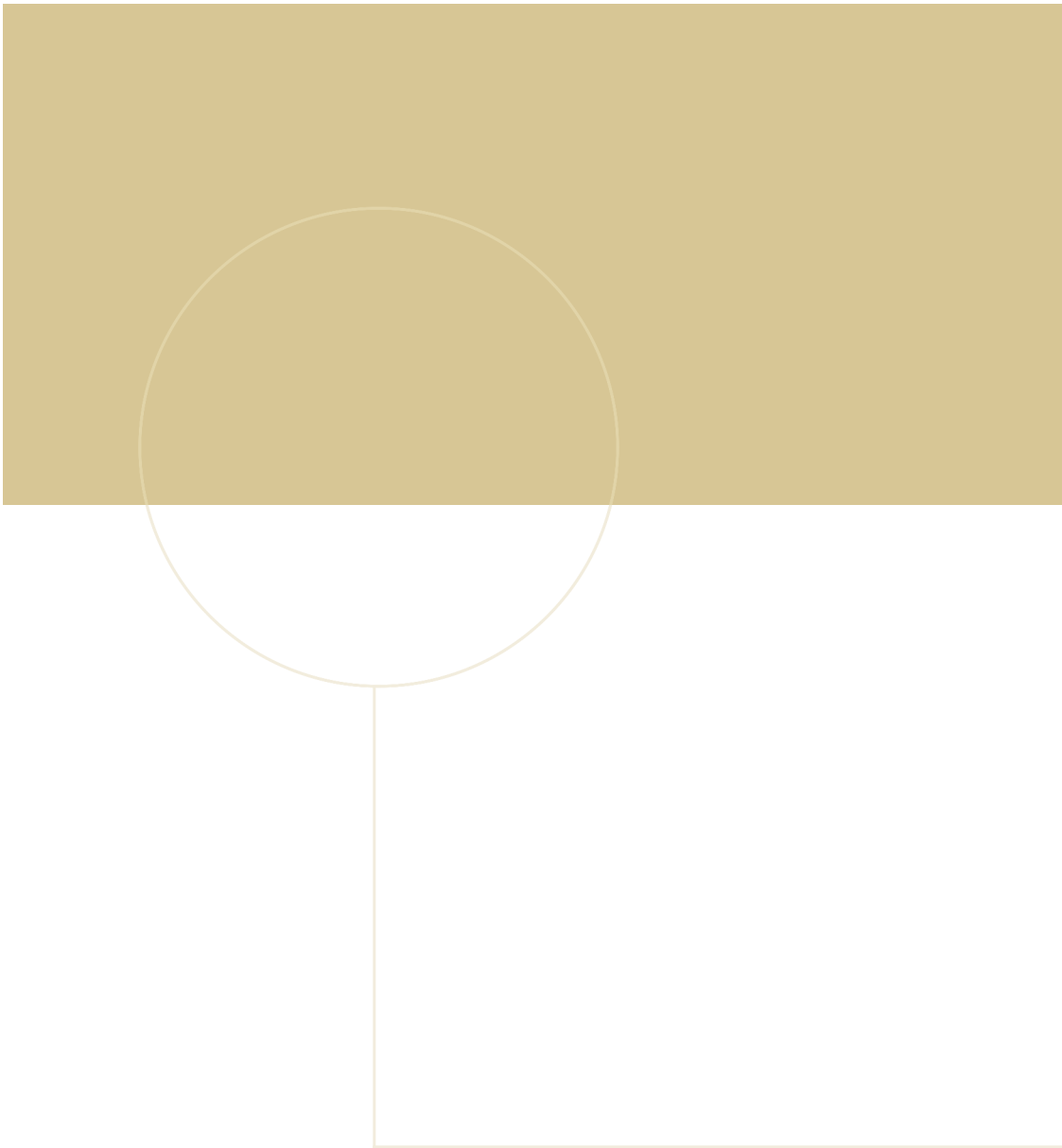


```
========================= RECORDING RESULTS ==================================
Total episodes: 17
-------------------------- Model info ------------------------------------
Time steps: [761, 823, 210, 564, 504, 2600, 732, 223, 2600, 2600, 224, 159, 643, 2400, 247, 2600, 164]
mean: 1062.0
Model number: ['25', '25', '25', '25', '25', '25', '25', '25', '25', '25', '25', '25', '25', '25', '25', '25', '25']
Model type: ['Temporal', 'Temporal', 'Temporal', 'Temporal', 'Temporal', 'Temporal', 'Temporal', 'Temporal', 'Temporal', 'Temporal', 'Temporal', 'Temporal', 'Temporal', 'Temporal', 'Temporal', 'Temporal',
'Temporal']
Epoch: ['30', '26', '23', '20', '19', '16', '13', '10', '09', '08', '07', '06', '05', '04', '03', '02', '01']
Loss: ['0.087', '0.092', '0.097', '0.099', '0.102', '0.108', '0.112', '0.119', '0.133', '0.151', '0.161', '0.184', '0.257', '0.262', '0.312', '0.336', '0.395']

-------------------------- Driving scores ------------------------------------
Steering score: [3.49, 6.01, 1.71, 5.45, 4.04, 7.13, 4.1, 2.98, 11.23, 8.05, 2.71, 2.9, 4.43, 2.34, 3.35, 1.5, 1.31]
mean: 4.278235294117646
Speed score: [16.25, 7.63, 19.15, 11.77, 12.18, 28.2, 15.27, 19.39, 28.2, 28.19, 18.58, 20.15, 8.75, 29.47, 16.03, 28.45, 20.32]
mean: 19.29294117647058
Speed below score: [17.75, 8.59, 24.36, 13.01, 13.71, 28.2, 15.09, 23.3, 28.19, 28.18, 23.93, 20.15, 9.45, 29.47, 21.79, 28.45, 20.28]
mean: 20.817647058823533
Speed above score: [13.07, 3.44, 14.89, 5.73, 4.64, 0.0, 15.55, 16.02, 0.0, 0.0, 14.27, 0.0, 4.71, 0.0, 10.49, 0.0, 0.0]
mean: 6.047647058823529
Count below speed_limit: [1389, 1689, 283, 1199, 1071, 7364, 1237, 300, 7399, 7368, 291, 457, 1337, 6496, 351, 7175, 480]
mean: 2699.1764705882235
Count above speed_limit: [664, 390, 355, 252, 220, 1, 654, 361, 1, 1, 370, 2, 234, 2, 371, 1, 1]
mean: 228.23529411764707
Crossed broke line: [5, 0, 0, 2, 2, 0, 4, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0]
mean: 0.8823529411764706
Crossed "none" line: [9, 1, 6, 2, 5, 0, 10, 4, 0, 0, 5, 2, 10, 2, 4, 0, 4]
mean: 3.7647058823529412
crossed other lines: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
mean: 0.0
target_reached_count: 1
target_reached_epochs: ['26']
target_reached_folder: [1]
target_reached_steer: [6.01]
target_reached_speed: [7.63]
collision_count: 11
not_moving_count: 5
```

Figure 20: Results from testing of model 25 during exploitation phase

B Complete results from final testing



NTNU
Norwegian University of
Science and Technology