Vehicle trajectory prediction using recurrent LSTM neural networks

Master's thesis in Complex Adaptive Systems

ALEXANDER BÜKK

RICKARD JOHANSSON

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2018:NN

Vehicle trajectory prediction using recurrent LSTM neural networks

ALEXANDER BÜKK
RICKARD JOHANSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2020

Vehicle trajectory prediction using recurrent LSTM neural networks

ALEXANDER BÜKK
RICKARD JOHANSSON

Supervisor: Anders Ödblom, Volvo Car Corporation
Examiner: Jonas Sjöberg, professor at the department of Electrical Engineering

Master's Thesis 2020
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Flow chart describing the neural network model developed in this project.

Typeset in LaTeX
Gothenburg, Sweden 2020

Vehicle trajectory prediction using recurrent LSTM neural networks

ALEXANDER BÜKK
RICKARD JOHANSSON
Department of Electrical Engineering
Chalmers University of Technology

Abstract

A Long short-term memory (LSTM) recurrent neural network (RNN) model was trained on making vehicle trajectory predictions based on real world driving data at a minimum driving speed of 70 kph. It was found that the trained model predict future trajectories with an root mean square error (RMSE) of $0.66[m]$ for a dataset containing two lane roads, and $0.63[m]$ for a dataset containing three lane roads, where a prediction is made up of 21 points covering five seconds of future driving. Multiple models were also trained jointly to generate multiple prototype trajectories, where the goal was to make each prototype trajectory specialize on different types of traffic scenarios and driving maneuvers. Another model was also trained on deciding which prototype trajectory of the jointly trained models would yield the lowest prediction errors. It was found that none of the implemented models learned to utilize the surrounding object information in the training data to take interactions with other road users into consideration when making trajectory predictions, which would be an interesting research topic for future work.

Keywords: Microscopic Traffic Flow Simulation, Trajectory Prediction, Deep Learning, Machine Learning, Artificial Neural Network, LSTM, RNN.

Acknowledgements

We want to thank our supervisor Anders Ödblom at Volvo Cars Corporation and our examiner Jonas Sjöberg at Chalmers University of Technology.

Alexander Bükk & Rickard Johansson,
Gothenburg, August 2020

Contents

List of Figures

List of Tables

1

Introduction

The research areas of Advanced Driver Assistance Systems (ADAS) and Autonomous Driving (AD) are under rapid development and the most pressing challenge is to ensure safety of drivers, passengers and other road users to decrease traffic fatalities [1, 3, 4, 5]. In 2016, 1.35 million people were killed in traffic related accidents world wide [6]; a number which is steadily increasing. ADAS and AD systems need to be able to anticipate other road users' intentions so that accidents proactively can be avoided [1, 5]; in other words, they need to be able to predict the future trajectories of surrounding objects and plan how to adapt accordingly. This thesis will in light of this investigate trajectory predictions, which can be used to model drivers' or other road users' driving behaviours.

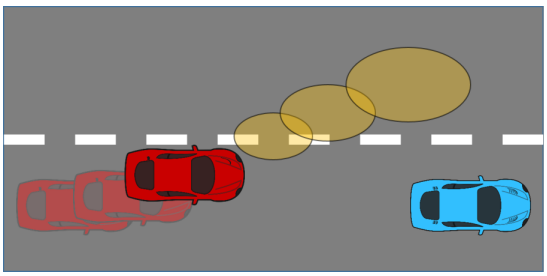
1.1 Background

Trajectory prediction is an integral part of ADAS and AD systems. Such systems use perceived knowledge of their environment to plan safe trajectories for the ego vehicle. Unfortunately, trajectory prediction is a difficult task due to the infinite search space of possible trajectories to choose from. The difficulty is not only due to the many possible traffic scenarios that can arise from different environments and surrounding objects, but also due to the drivers' intentions and habits being non deterministic [3]. Although some trajectories can be invalidated due to physical limitations on vehicles and drivers, the subset of trajectories which would violate traffic rules must be taken under consideration since all drivers do not follow all traffic rules. All these reasons points to trajectory predictions being a complex task where the problem is to find methods of predicting the most probable trajectory of a vehicle in a given scenario.

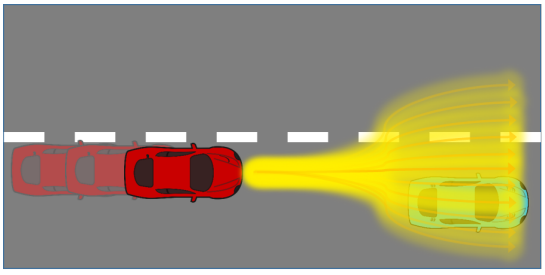
Different approaches for vehicle motion and trajectory prediction have been proposed in the literature, and a comprehensive survey can be found in [4]. In this survey, the different models are divided into three families: Physics-based, maneuver-based and interaction aware motion models. In physics-based motion models the vehicle is represented as a physical entity, where the evolution of the vehicle's future

states are modeled by dynamic [7, 8, 9] or kinematic [10, 11, 12] equations. These models are extensively used and are good for short-term predictions, but do not perform well for long-term predictions [4, 5]. The maneuver-based motion models do in turn represent vehicles as independent maneuvering (or behaving) entities where previous maneuvers are used for prediction of future, matching maneuvers. Unfortunately, traffic scenes involving more than one intelligent maneuvering entity, such as road intersections, are not modelled very well [4]. Finally, interaction-aware motion models aim to solve the shortcoming of the previously mentioned model by modelling vehicles as maneuvering entities which interact with other maneuvering entities. This last type of model is the most comprehensive family of models: they enable for longer-time predictions than physics-based models, and are more reliable than maneuver-based models, due to the account of inter-dependencies between objects [4].

Techniques used for trajectory prediction by the different types of models vary much in computational complexity and how long the trajectories are valid. Physics-based models are good in real-time systems since they are very computationally efficient and operate under the assumption that the states of a vehicle are completely known, as well as the evolution of its future states [4]. However, since there are both uncertainties in the current state of a vehicle and it's evolution, this type of model can only make valid predictions for trajectories up to one second into the future [4]. These uncertainties are commonly modeled by normal distributions using Kalman Filters [13] for state estimations, from measurements from multiple sensors (sensor fusion), which is a special case of Bayesian filtering [4]. Trajectory predictions from these distributions are commonly made by Gaussian noise simulations (see figure 1.1a) or Monte Carlo simulations [14] (see figure 1.1b) for some time steps into the future.



(a) Gaussian noise simulation with uncertainties for some time steps into the future. Predictions are made on the states of the red car, where uncertainties are represented by the yellow ellipses.



(b) Monte Carlo simulation for some time steps into the future. Predictions are made on the states of the red car, where sampled paths of varying probability are represented by yellow arrows.

Figure 1.1: Uncertainties of a vehicle's states are in physics-based models commonly modeled with Gaussian noise (a) or Monte Carlo simulations (b).

Maneuver-based model trajectory predictions are either based on prototype trajectories, or driver intention estimations [4]. They utilizes methods such as as Gaussian Processes [15] and mixture models, dynamic Bayesian networks and coupled hidden

Markov models. These new approaches have shown higher accuracy compared to the physics-based models, but lack in efficiency and need much engineering and fine tuning by humans [4].

Interaction-aware motion models are scarcely represented in literature, but are commonly based on prototype trajectories or Dynamic Bayesian Networks [4]. This type of model require more computational power than the other models, due to its additional complexity, but is also the most accurate one [4]. Recent increase in computational power, due to advances and use of fast Graphical Processing Units (GPUs), has opened a new avenue of possibilities by introducing deep learning approaches to these problems. Deep learning has successfully been used in solving tasks such as object tracking and semantic road segmentation, and is now also researched in the context of building driver models. It is believed that deep learning could be used to generalized and invariant learning, which could decrease much of previous needed hand engineering. Development of Deep Neural Networks (DNNs), such as Recurrent Neural Networks (RNNs), and Long Short-Term Memory networks (LSTMs) in particular, has shown great results in learning complex, time-dependent relationships and will in this thesis be investigated in the context of making short to long-term vehicle trajectory predictions.

1.2 Goal

In this thesis, we explore a supervised deep learning approach to vehicle trajectory prediction, that can be used used when modeling driving behaviors. The goal is to design a model which takes sequential samples of logged ego vehicle sensor data as input and outputs trajectory predictions for the ego vehicle's future path (see figure 1.2). This thesis attempts to quantify potential gains of utilizing LSTM cells in an RNN to achieve accurate medium to long-term vehicle trajectory predictions. The model is implemented with the artificial neural network (ANN) framework Pytorch [16] and utilizes Nvidia's CUDA accelerator [17] for GPU computation. Essential challenges are:

- establishing Pytorch framework environment for machine learning;
- data transformation and pre-processing;
- implementing an RNN trajectory prediction model with LSTM cells;
- training and evaluating the model; and
- compare prediction errors to contemporary results.

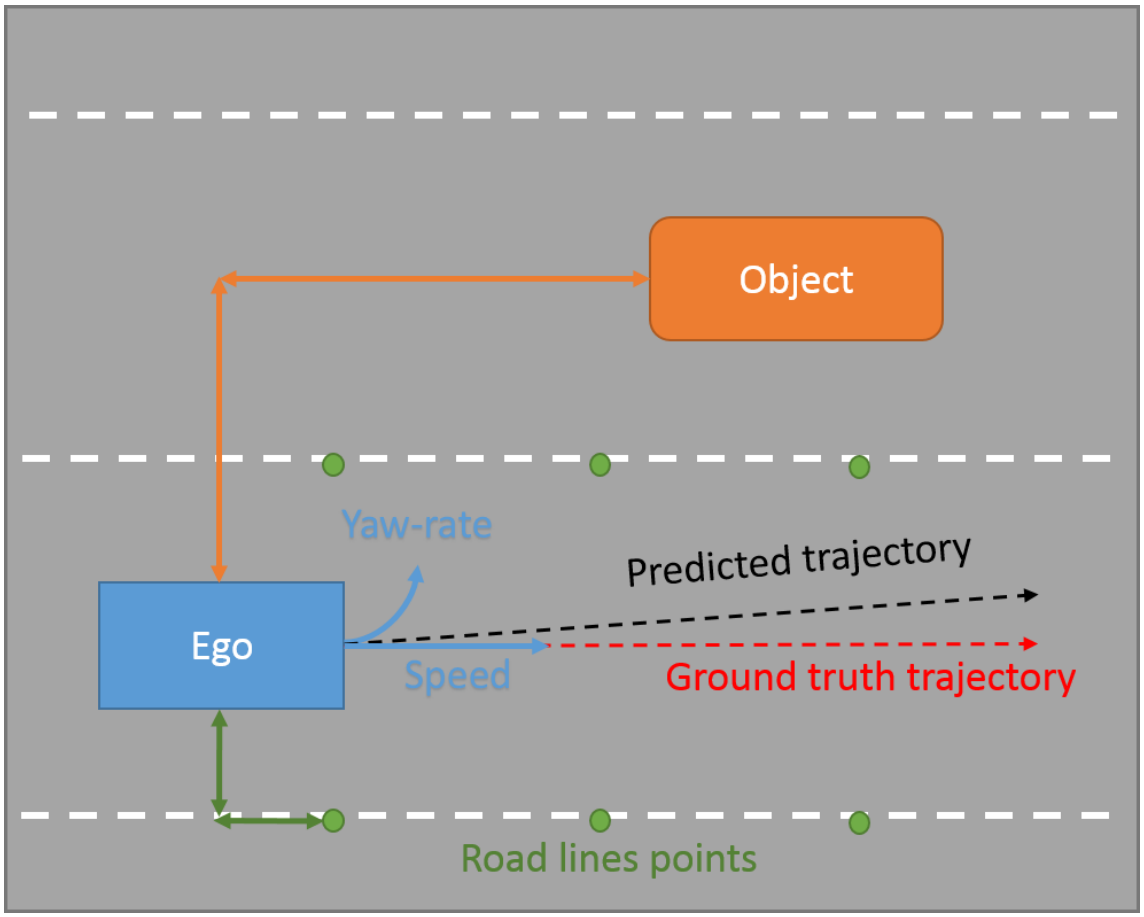


Figure 1.2: Problem description over trajectory prediction. The goal is to make a trajectory prediction as close to the ground truth trajectory as possible. The inputs used as basis for predictions are: ego position and yaw rate; sensed objects' coordinates; and road line marker estimates.

1.3 Scope

The Scope is to investigate deep learning with LSTM for vehicle trajectory prediction using training data consisting of approximately 50 hours and $[2,000[km]$ of logged car expedition sensor data. The thesis work is limited to considering traffic scenarios where cars are driving with a minimum speed of 70 km/h, and where at least one lane change per log data file was found, where each file consists of approximately 80 seconds of driving. The thesis does not aim to predict when lane changes should be made, even though an analysis is provided on the predicted trajectories tendency to end up in same lane as the ground truth trajectories. Nor is the thesis aimed at filtering out only feasible trajectory predictions; the model simply makes trajectory predictions based on statistical fitting on the provided training data. The thesis does not consider how the model's predictions could be implemented in an ADAS or AD system, or a simulation environment: it simply concerns trajectory predictions.

1.4 Workflow

The work that this thesis is based on utilized LSTM RNNs for trajectory prediction. After some initial research on the trajectory prediction literature, it was decided that we should investigate further into how LSTM DNNs can be used for trajectory predictions, with the aim of improving the accuracy of the prediction, compared to the previous project. The framework that was used in the previous project for implementing LSTM RNNs was PyTorch. Since this framework is one of the more popular and beginner friendly machine learning (ML) frameworks for designing LSTM networks, it was decided that we would use the same framework and invest some time into learning how to use it. After that, an ML environment was setup so that we could train and evaluate our LSTM model implementation. In parallel to the model evaluation, we iteratively analyzed the available training data, and much time was devoted on selection and pre-processing of the data, with the aim of increasing the accuracy of the predictions by using higher quality data. And finally, statistical metrics where calculated, such as the distributions of the longitudinal and lateral trajectory prediction errors. We did not only look at the prediction errors, we also analyzed if the predicted trajectories were laid out in the same lanes as the ground truth trajectories. The work was by no means linear, in the sense that all data was first collected, then pre-processed, followed by model selection, training and validation, and finally ending with an analysis of the trajectories; the work was going back and forth over many iterations. However, this will not affect the structure of the report, since we only report on our final findings.

1.5 Outline

The thesis is divided into six chapters: Introduction, theory, methods, results, discussion, and conclusion. A theoretical basis is provided for understanding of concepts and frameworks necessary for understanding the work done in the thesis. The methods section describes how the theory was implemented and thereafter, the results are presented and compared to related articles results. A discussion is then held concerning the validity of our results and alternative approaches are proposed. The last section concludes our findings and suggests further research to be conducted on the topic of this thesis.

2

Theory

In this section we introduce neural networks and recurrent neural networks (RNN) with focus on the Long-Short-Term Memory (LSTM) RNN. We explain the basics behind how they work and also which type of problems they are good at solving. All necessary theory to comprehend this thesis work will be covered here.

2.1 Statistical learning

Statistical learning is the combined field of statistics and machine learning and dates back to the 1960's. Until the 1990's, statistical learning was considered a theoretical field since earlier statistical learning algorithms were too computationally heavy for computers of that time. However, new algorithms and faster computers eventually enabled for utilizing the theoretic knowledge in practice for estimating multidimensional functions. Some unique and popular problems that statistical learning has being used for are probability distribution estimation, pattern recognition and regression estimation.

The goal of statistical learning is to find a predictive function based on previously observed data. It is assumed that such a function can be estimated over a distribution $\mathrm{p}(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ is the explanatory variable and $\mathbf{y}$ is the response variable; that is, the goal is to find the function $f(\mathbf{x})$ that best predicts $\mathbf{y}$. The process of finding this function is called *training* or *learning* [18]. There are two types of statistical learning: *supervised* learning and *unsupervised* learning. We will only be concerned about the supervised type, since that is the one used in this thesis. The function f in this study will be expressed in the form of an ANN, which will be explained in detail in section 2.2.

2.1.1 Supervised Learning

Supervised learning is the method of learning a function f to map an input $\mathbf{x}$ to a prediction of the output $\mathbf{y}$ [18]. The function will learn from labeled data consisting of pairs of inputs $\mathbf{x}_i$ and outputs $\mathbf{y}_i$ from a training set. The training set contains training examples $((\mathbf{x}_1, \mathbf{y}_1), \ldots (\mathbf{x}_n, \mathbf{y}_n))$. A model which aim to approximate the function f will have its parameters updated through a training process, using the training data. The goal is to be able to use the model for mapping new, unseen examples $\mathbf{x}_i$ to new prediction $\hat{\mathbf{y}}_i$. To test how well the learned mapping predicts new data, the model is tested on a separate set called test data which is a separate data set excluded from the training set. This mapping comes in two forms, described below.

Regression: Given an observation $\mathbf{x}$, regression is the task of finding a corresponding value $\mathbf{y}$.

Classification: Given an observation $\mathbf{x}$, classification is the task of finding which of a set of categories $\mathbf{x}$ belongs to.

2.1.2 Objective Function

In statistical learning, the training of a function approximator can be formulated as an optimization problem, where the goal is to minimize or maximize an objective function. In the case of supervised learning, the goal is to find a function, in a function space, which maps x onto y with as small an error as possible. One very common measure of the error of the approximated function is the mean squared error (MSE),

$$\mathrm{MSE(Y)} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2, \tag{2.1}$$

where $\hat{Y}$ is a vector of predictions on X.

2.1.3 Optimization Methods

The task at hand is to learn a model a certain task. To learn a task there must first be a quantitative preference measurement for how well the model performs the task. This is usually measured by an error function $E(f_{\boldsymbol{\theta}}(\mathbf{x}),\ \mathbf{y})$, which produces a value on the error between the prediction $f_{\boldsymbol{\theta}}(\mathbf{x})$ and the target value $\mathbf{y}$. By updating the parameters $\boldsymbol{\theta}$ of the model to minimize the error, the model will learn the task it has been given.

One method frequently used for optimizing is the gradient descent method. By updating the parameters of a model by small increments in the negative direction of the gradient one can iteratively step down towards a local minimum of the objective function. To be able to calculate the gradient of an error function E, the function must be differentiable. The update formula for a model's parameters $\boldsymbol{\theta}$ will then be:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \gamma \nabla_{\boldsymbol{\theta}} E(f_{\boldsymbol{\theta}}(\mathbf{x}),\ \mathbf{y}), \tag{2.2}$$

where γ is the learning rate. With a small γ, the learning will be more precise and steady, but slow. With a large γ, the learning will be fast but more unstable. As an analogy, a minimization problem could visually be thought of as the problem of finding the global minimum in a landscape of many craters of varying depth. If the learning rate of the optimization method is to low, the search for a solution would be limited to the first crater the algorithm starts looking inside. The solution would then be a local minimum, but may never reach potentially lower minima in nearby craters. However, if the learning rate is too high, the solution candidate will jump between different craters and may never reach a minima. Therefore, the learning rate needs to be within an adequate range to avoid inefficiency and poor results. A common practice is to start an optimization with a high learning rate and successively lower it over its iterations. Several other optimization methods that are commonly used in machine learning are based on the same theory as the gradient descent method, but are improved further to suit different types of problems. One such method is called ADAM which is commonly used for parameter optimization in deep learning [19].

2.1.4 Generalization

When training a predictive model on some training set, the goal of the model is to make accurate predictions on new data, which the model has not been trained on. If the difference is small between the errors of the predictions of the training data and the test data, then the model is said to have generalized well. However, if the difference is large, the model has not generalized well. A common reason for a model not generalizing well is that the model becomes overfitted on the training data, i.e. becomes really good at making prediction only on the training data, but not on new (test) data. The causes for overfitting a model could be many, and involves the parameters and complexity of the model, as well as the training method. To prevent low generalization, methods for preventing this has been developed and belong to the theoretic toolbox named Regularization, which will be reviewed in the following section.

2.1.5 Regularization

In machine learning problems, a major problem that arises is that of overfitting. Because learning is a prediction problem, the goal is not to find a function that most

closely fits the previously observed data, but to find one that will most accurately predict output from future input. Empirical risk minimization runs this risk of overfitting: finding a function that matches the data exactly but does not predict future output well. Overfitting is symptomatic of unstable solutions; a small perturbation in the training dataset would cause a large variation in the learned function. It can be shown that if the stability for the solution can be guaranteed, generalization and consistency are guaranteed as well [20, 21].

2.1.6 Hyper Parameter grid search

When training a statistical model, there are usually several parameters that will affect the learning outcome: both in how fast the models learns the desired properties and how precise the predictions become. These parameters are not learned through the training algorithm and must be defined before training. These parameters are called *hyper parameters*, and are distinguished from the internal parameters of a statistical model. The learning rate of a model or the size of a artificial neural network are examples of hyper parameters. The optimal values of the hyper parameters are hard, or even computationally impossible to find, due to the vast search space of combinations of different hyper parameter values. Additionally, the fact that a model can have varying optimal hyper parameter values, depending on the training data itself, makes this task even harder. However, to find reasonable values for these parameters, it is common to run parameter sweeps over all parameters, where adequate, individually specified ranges are chosen for each parameter. The process of performing these sweeps are called *grid search*, since the different parameter ranges together could be thought of as a multidimensional grid being searched for the optimal position. A large grid space could be used in the early search process, and later on a smaller grid space around the previously found best position can be used to improve the parameter tunings even further. Since the training procedure of a model can take several hours, or in some cases days, a grid search that finds the optimal solution is too time consuming, which is why this method often becomes more of an art than a strictly scientific method.

2.1.7 Cross-Validation

Cross-validation is a statistical technique used to generalize and validate the performance of a model. Cross-validation is mainly used in cases where the goal of the task is any kind of prediction. This is done by dividing the data into two separate subsets: training and validation. The training set is used for training the model and updating the parameters of the model. The validation set is used after one round of training to see how well the model generalize and if the training is overfitting. In the early stages of training the prediction error of both the training and validation sets should decrease. Eventually as the model starts to learn the specific characteristics

of the training set the prediction of the validation set starts to decrease (the error increases). It is important to stop the training when the validation prediction error is the lowest because that is where the models peaks both in performance and generalization. Stopping the training when the validation prediction error is the lowest is called *early stopping*.

2.1.8 Variable importance

Variable importance (VI) is a measurement, performed on each feature (variable) in the data separately, of the impact a certain variable has on the output of the model. One way of measuring VI is by first permuting one variable among all data points, then feed the altered data to the model, calculate the prediction error and last compare this prediction to the prediction error of the original data. The variable causing the largest error will have the highest importance since changing this variable gives the largest error. This method is useful when selecting which features to use when training a statistical model.

A simple example to clarify the permutation process is shown below in matrices A, A_x, A_y and A_z. Here A represents the original data where each row represents one input sample and each column represents a variable. Each matrix A_i, where $i = x, y, z$, shows the data matrix A where variable i has been permuted randomly.

$$A = \begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{pmatrix}, \quad A_x = \begin{pmatrix} x_2 & y_1 & z_1 \\ x_1 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{pmatrix}, \tag{2.3}$$

$$A_y = \begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_3 & z_2 \\ x_3 & y_2 & z_3 \end{pmatrix}, \quad A_z = \begin{pmatrix} x_1 & y_1 & z_3 \\ x_2 & y_2 & z_1 \\ x_3 & y_3 & z_2 \end{pmatrix} \tag{2.4}$$

2.2 Artificial Neural Networks

One type of statistical learning model is the artificial neural network (ANN). It is inspired by biological neural networks in animal brains, and vaguely imitates their behaviour and structure. In an ANN, artificial neurons are connected in a network and serves as computing units. Each neuron can be described as a nonlinear function which takes multiple weighted inputs, sums them up and calculate an output based on the choice of *activation function*. In biological neural network, the activation function simply compare if the weighted sum is above a certain threshold, and then gives a binary output: 'fire' or 'do nothing'; however, many activation functions used

in artificial neurons often outputs real values. These neurons can when connected in a single network layer be used for arbitrarily well approximating a wide range of continuous functions, according to the *universal approximation theorem* [22]. A catch though is that the theorem does not state how many neurons are needed nor which activation function or training method should be used when designing the network. In the next sections, we will introduce the simplest type of ANN, the *Perceptron*; give an overview of multi-layered networks; and elaborate on how they can be trained by sampled data to predict the response of an observed variable.

2.2.1 Perceptrons

The fundamental building block of an ANN is the perceptron which consists of one single neuron. The perceptron is described mathematically in equation 2.5:

$$y = \phi\left(\sum_{j=1}^{n} \omega_j x_j + b\right), \quad \phi(x) = \begin{cases} 1, & \text{if} \quad x > 0 \\ 0, & \text{otherwise} \end{cases}, \tag{2.5}$$

where the sum is taken over each element x_j of a real valued input vector, multiplied by the corresponding element ω_j of real valued weight vector. Thereafter, a bias b is added to the sum and the new value is then passed as input to an activation function $\phi(\cdot)$, which returns one if the input is greater than zero and otherwise zero, which becomes the output y of the neuron. The schematic representation of the same perceptron is depicted in Figure 2.1.

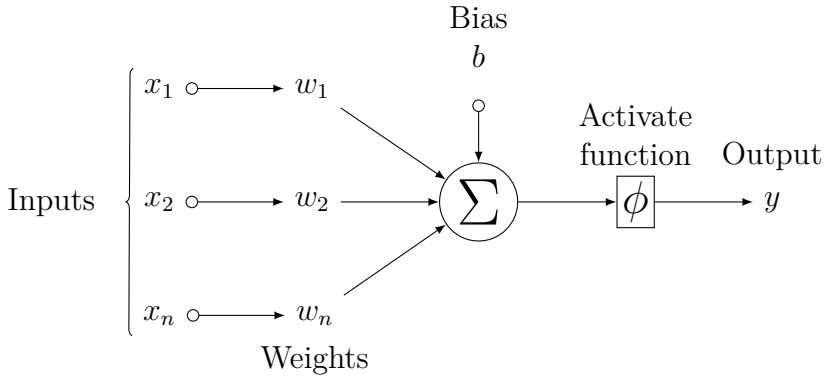


Figure 2.1: Diagram of a simple perceptron with a single input layer.

A single perceptron is not very useful for approximating complicated functions. Luckily, they can be used for building more complex structures, which will be discussed in the next section.

2.2.2 Multi-layer Neural Network

A multi-layered neural network expands on the theory of the simple perceptron in the previous section by utilizing many neurons, connected to each other and structured in layers as depicted in Figure 2.2. The neurons and their connections are represented by circles and edges, respectively. The output of each neuron is calculated similarly as for the perception, as can be seen in Equation 2.6.

$$\boldsymbol{y}^{(l)} = \phi\left(\boldsymbol{W}^{(l)}\boldsymbol{y}^{(l-1)} + \boldsymbol{b}^{(l)}\right) \tag{2.6}$$

For each layer l, $\boldsymbol{y}^{(l)}$ is the output vector, $\boldsymbol{W}^{(l)}$ is the weight matrix for all connections between layers l and $l-1$, $\boldsymbol{y}^{(l-1)}$ is the input vector, and $\boldsymbol{b}^{(l)}$ is the weight bias. By using multiple layers and neurons it is possible to train a network to approximate much more complex functions than the single neuron perceptron.

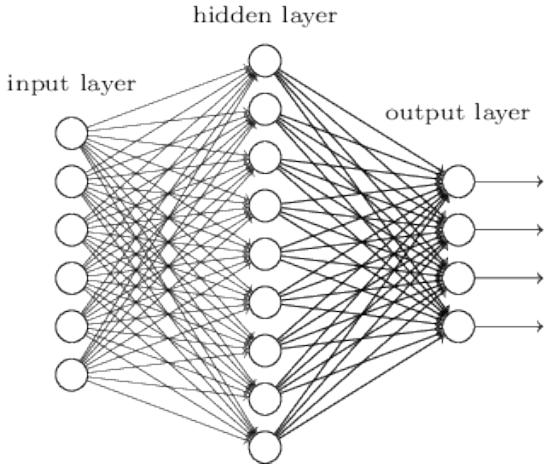


Figure 2.2: Fully connected ANN with an input layer, one hidden layer and an output layer. `https://qph.fs.quoracdn.net/main-qimg-337f88c13646d861e422d1246e201071`

2.2.3 Activation Functions

Activation functions are used inside the neurons of ANNs to transform the induced local field v,

$$v \equiv \sum_{j=1}^{n} \omega_j x_j + b, \tag{2.7}$$

to adequate outputs. For instance, the property of non-linearity can be introduced to a network by using nonlinear activation functions. Activation functions also serves the purpose of squashing the induced local field to a certain interval, commonly between zero and one. This is done to keep the output of each neuron in the same

range. Usually, all the neurons in a network use the same activation function, except in the output layer. Listed below are some popular non-linear activation functions for the hidden layers.

Sigmoid

$$\phi_{sigmoid}(x) \equiv \sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.8}$$

Hyperbolic Tangent

$$\phi_{tanh} \equiv tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.9}$$

ReLU

$$\phi_{ReLU} = \max(0, x) \tag{2.10}$$

One important requirement which an activation function must fulfill is non-linearity. If a linear function would be used as activation function, a network of several layers would be equivalent to a linear transformation. Non-linearity gives a neural network much more capability to approximate complicated functions. In fact, for the universal approximation theorem (mentioned in section 2.2) to hold the activation function must be non-linear. Another property that activation functions must adhere to is differentiability. This property is crucial when training a neural network, as will be shown in the next section where the training algorithm for ANNs is described.

The activation function for the output layer depends on which type of output is desired since range of the output of an ANN needs to match the target values range. For regression it is common to use a linear activation function.

Linear

$$\phi_{linear} = cx \tag{2.11}$$

For classification where the output maps the probability of belonging to a certain class the softmax activation function is often used.

Softmax

$$\phi_{softmaxi} = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_i}} \tag{2.12}$$

where $\phi_{softmaxi}$ is the probability of belonging to class i.

2.2.4 Backpropagation

The most commonly used algorithm for training ANNs is called backpropagation. This algorithm uses the gradient decent optimization algorithm to minimize an

objective function. Backpropagation is most commonly used in supervised learning since it requires the derivative of the objective function, with respect to the output, to be known, which only can be computed if the desired output value is known. The algorithm has two stages: the forward stage and backward stage. During the forward stage, data propagates through the network and produces an output. The output is compared to the target value and the error is calculated. During the backward stage, the gradients of each weight in the network is calculated. Then, all weights are updated in the negative gradient direction by a small amount. The amount which the weight is updated by is dependent on the learning rate. With a large learning rate the network learns fast and with a small learning rate the learning is more accurate.

2.2.5 Weight Initialization

Before starting the training of an ANN, the network weights must be initialized. As mentioned in section 2.1.3, the gradient descent optimization methods descends down the steepest gradient of a function to find the local minimum. Unfortunately the function which is optimized when training an ANN is not convex, which means that the starting value of the model parameters matter. Depending on their initialization the optimization algorithm will find different minima. One naive way to initialize the weight would be to set all of them to zero. Unfortunately, setting all the weights to the same value will make the network unable to learn. This is due to the fact that the error backpropagated during learning is proportional to the weights of the network. Initializing all the weight to the same value will therefore make the backpropagated errors identical and accordingly all the weights, when updated in the same iteration, will be updated the same magnitude.

This is why weights are randomly initialized when creating a new ANN. The random initialization of the weights are sampled from some distribution, usually a Gaussian or a uniform distribution with zero mean and small variance to avoid large weight initialization.

Another way to initialize the parameters for an ANN is to use a pre-trained ANN which has completed training for another task and start the learning on the new task from there.

2.2.6 Vanishing and exploding gradient

A common issue when training neural networks, especially deep neural networks, is that of vanishing and exploding gradients. In a regular feed forward network, the gradients of the weights in the foremost layers of the network is dependent on the gradients and values of the weights in the succeeding layers of the network.

The gradients in the layers close to the input layer of the network are calculated by multiplying several of the gradients and values of the weights in the end of the network. If the absolute value of the gradients are below one these multiplications are in risk of resulting in a very small number and therefore make the gradient vanish. If the absolute value of the gradients are larger than one the multiplications runs the risk of giving a very large number and therefore explode. This is especially a problem in recurrent neural networks but can be solved by using Long Short-Term Memory Cells see section 2.3.1.

2.2.7 Data Normalization

Normalizing the data so that every variable has mean zero and unit variance generally speed up learning convergence. Shifting the variables so that they all have mean zero will give the network an unbiased update and therefore speed up training. This should therefore be done for all layers of the network. This is one reason why activation functions such as the sigmoid function is a popular choice since it basically automatically normalize the output of a layer. It is important that all variables are scaled the same. Scaling them all to unit variance is easy and standard practice. The actual performance of the network is not affected by the scaling the inputs but the learning is. If the inputs are very large or very small the weights are going to compensate for this and be very small or large. A model with very high values for the weights runs higher risk of being unstable resulting in bad performance during training and very sensitive to inputs which gives lower generalization.

2.2.8 Deep Learning

The definition of deep learning is simply an artificial neural network which consists of several hidden layers. There is no exact definition of how many layers are needed, but a network with two or more hidden layers are usually accepted as deep. An ANN consisting of only one hidden layer is called shallow. One benefit of having several layers is that each layer can learn different abstractions of the data. Exactly why deep networks perform better than shallow networks is unknown but empirical data have shown that deep networks perform better [23, 24, 25, 26, 27, 28]. One drawback though is that the computational complexity increases with deeper networks.

2.3 Recurrent Neural Networks

One problem with regular feed forward neural networks is how it handles sequential data where the current input and output is dependent of the previous inputs and outputs. It is however possible to feed in a fixed sequence with previous inputs

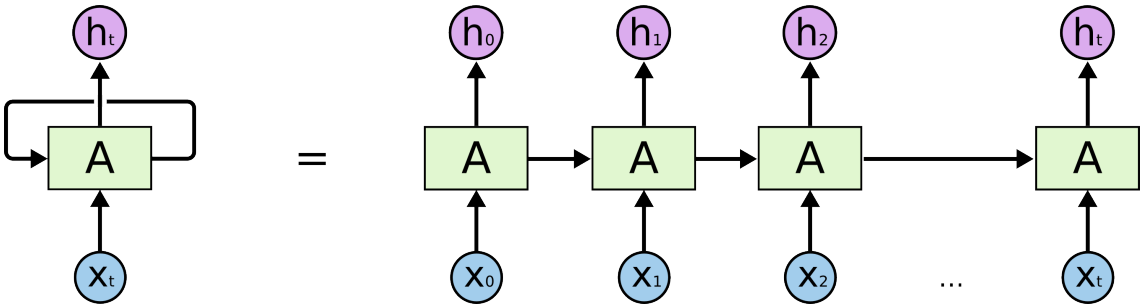


Figure 2.3: The repeating module in a standard RNN. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/RNN-unrolled.png`

into an ordinary feed forward ANN. This solution unfortunately makes the network inappropriate to use when the length of an input sequence is not constant. An example for when the input sequence is variable is when processing text, where words and sentences are of different lengths, and contextual patterns need to be learned. There is a special kind of network that solves such problems, and specializes in sequential data, that is called recurrent neural networks (RNNs). The main concept behind RNNs are that they are coupled with themselves, see Figure 2.3. The advantage of RNNs is that they make it possible to learn sequential structures such as time dependencies, which regular ANNs have difficulties with. The connection that links the network back to itself gives the network a feature comparable to memory. RNNs are also useful in modeling and analyzing sound and video since they stretch over time.

In theory, a regular RNN's memory should be both long-term and short-term, but in reality this is rarely the case. The biggest problem is the long-term memory: the ability to remember things that occurred several time steps earlier. The problem is analogous to the vanishing gradient problem in deep feed forward RNNs. The vanishing gradient problem occurs in RNNs even with just one layer. This is due to RNNs being connected to themselves recursively, and thus depending on information processed several steps before. Just as a layer in a feed forward network is dependent on later layers in the network, a recursive step in an RNN is dependent on later time steps. The gradients of later time steps will be small because they are calculated by multiplying gradients from earlier time steps, which will have an absolute value lower than one. This makes it hard to have an RNN that captures long term dependencies.

2.3.1 Long Short-Term Memory Cells

In 1997, Sepp Hochreiter and Jürgen Schmidhuber [29] introduced a RNN called long short-term memory (LSTM) which combat the vanishing gradient problem and enables long term memory as well as short term, hence the name. LSTMs makes long term memory possible by preventing the problem of vanishing or exploding gradients. LSTMs solves this by making every recurrent node a memory cell, instead of a node with an activation function. An LSTM does not only take in its previous state and

current input as ordinary RNNs, it also takes in the old state of the memory cells to calculate its output, see Figure 2.4.

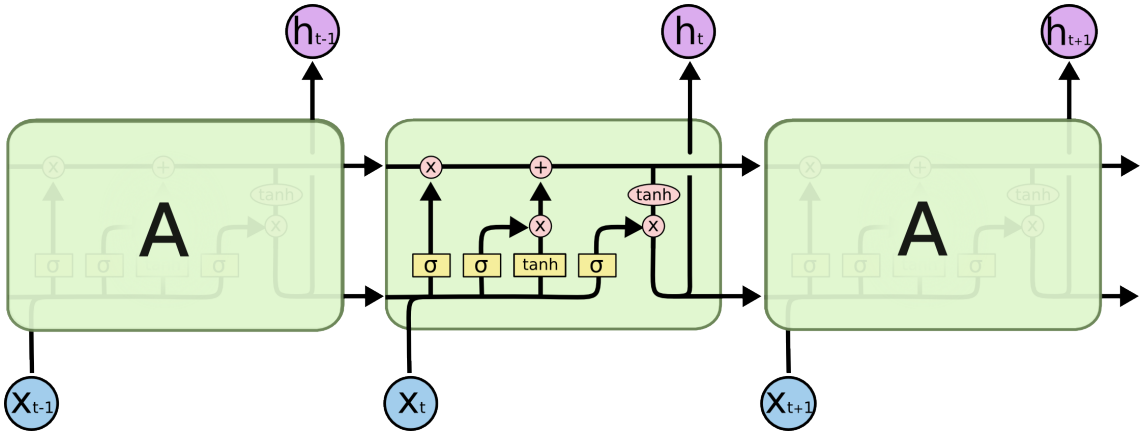


Figure 2.4: The repeating module in an LSTM containing four interacting networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-chain.png`

Basically, an LSTM cell has a memory, called the cell state in the form of a vector of real numbers, which is updated by four different feed forward ANNs. The cell state and the current input is what determines the current output of the LSTM cell. The four feed forward ANNs in the LSTM are designed for different tasks. The first decides what to keep in the cell state—what to remember. The next two networks decides what to add to the cell state—what to add to the memory from the current state. The last networks task is to decide what the LSTM cell should output in the current step. The exact relationships within an LSTM are shown in equations (2.13-2.18), Where c_t and h_t are the cell state and the hidden state of the LSTM cell at time step t.

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \tag{2.13}$$
$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \tag{2.14}$$
$$g_t = tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \tag{2.15}$$
$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \tag{2.16}$$
$$c_t = f_t c_{(t-1)} + i_t g_t \tag{2.17}$$
$$h_t = o_t tanh(c_t) \tag{2.18}$$

The dimensionality of the cell state and the hidden state are the same and is referred to as the number of hidden units of the LSTM cell. The hidden state h_t is what the LSTM cell outputs at each time t.

2.3.2 Input

The input to an LSTM network is a sequence of samples. Usually this would be a time-series of samples where each sample is data sampled at that exact time or data representing that time step. Several samples in a consecutive order builds up sequence which is used as input to the LSTM network.

2.3.2.1 Mini-batch

When training any type of neural network, using the errors of every output at the same time to update the weights corresponds to taking the steepest step within the parameter space and fastest convergence. This is called Batch Gradient Descent (BGD). This does not mean that it will converge at the global minima but just the closest minima to the starting position. A common way around walking down the closest local minima is to use Stochastic Gradient Descent (SGD). When using SGD the weights are updated after calculating the output error of only one output. This will add stochasticity to the walk through parameter space which means that it is possible to jump out of a local minima. Using only one output to update the weights can lead to too much stochasticity where converging takes too long time if it converges at all. Mini-batch Gradient Descent is a mix between BGD and SGD where the number of outputs used for updating the weights can range between 1 and n. This method can therefore utilize the benefits of both BGD and SGD with both stochasticity and fast convergence. While utilizing both they are still forced to be balanced. In the case of LSTM networks the mini-batch input would consist of several random sequences ranging between 1 and n. The sequences in a mini-batch should not be consecutive series. The number of sequences in a mini-batch is referred to as batch size.

2.3.3 Output Layer

The output of an LSTM cell will always be a vector of the same size as the cell state and with values between negative one and one due to the squashing property of the hyperbolic tangent function (see Equation 2.9). This is rarely the final desired output, which is why a simple feed forward ANN is used to transform the LSTM cell output to a desired form. The desired form could be continuous in one dimension for a regression problem or probabilities of belonging to different categories in a classification problem.

3

Methods

This chapter describes the methodology involved in learning a model to map sequences of traffic scenario samples to a trajectory forecast. We begin by exploring the data that was used for training the model, where the filtering and pre-processing of the data is described as well as how the data was split into training and validation sets. Continuing, the network architecture of the proposed model is laid out in detail in terms of its constituting components. Also, the training procedure for the model is described with regards to the loss function, parameter optimization, and choice of hyperparameters. A final section is devoted to evaluation of the model, where evaluation methods and performance metrics are established.

3.1 Datasets

The model will be trained on sequences of features built up of samples from real data logged from sensors during driving. The logged data consists of about 6500 recorded files where each file contains roughly 80 seconds of driving—re-sampled to 4 Hz—on different kinds of roads. Input features comprise:

- ego vehicle yaw rate and velocity;
- surrounding objects positions relative to ego vehicle;
- estimated road information.

When creating the dataset used for training the trajectory prediction model, many log data files were filtered out due to requirements on lane markings visibility, number of identified highway lanes, and minimum speed of the ego vehicle. The log data files also underwent preprocessing in many steps before finally being divided into training and validation sets. These steps firstly involved re-ordering lane markings fields, filling in missing lane markings data and re-sampling of logged data. Then, input-target pairs was created from the log data. The input was taken as sequences of logged data samples, which had been extended with future lane markings data and was then normalized to zero mean and unit variance. The target contained the future path of the ego vehicle, extracted from future samples of the logged data and then converted to Cartesian coordinates. All of these analyses and manipulations

of the logged data are depicted as a flow chart in Figure 3.1 and will be described in detail in the following sections.

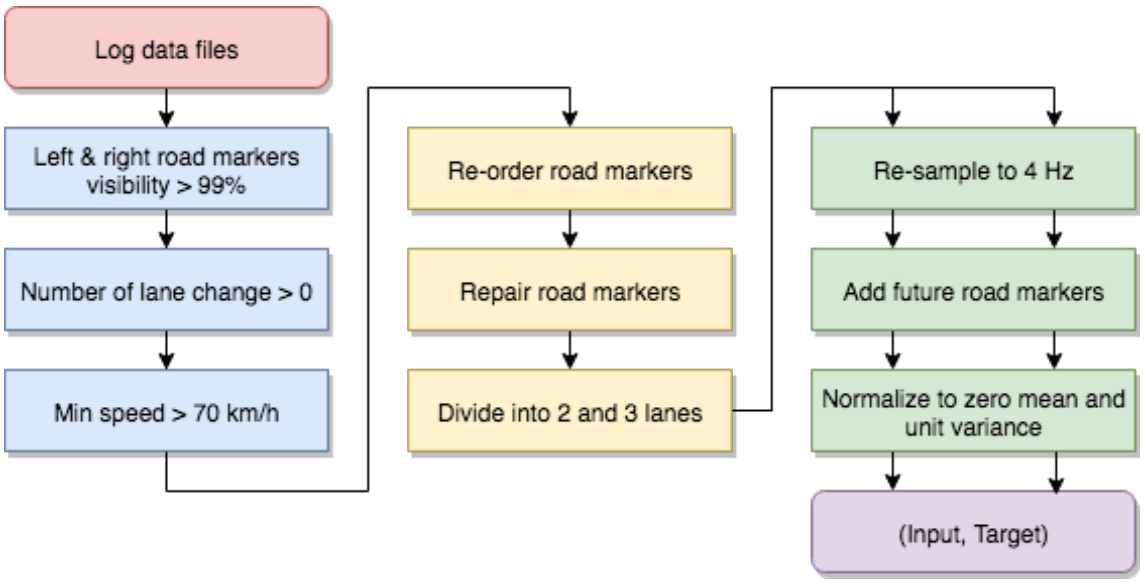


Figure 3.1: Flow chart of reprocessing of road data. Repair lane markers means filling in a few instances where a lane marker is missing. Add future lane markers means adding a whole new lane marker after discovering it (driving in one of the outer lanes on a road with three lanes means that the sensors will not discover the other outer most lane until a lane change has been done. That lane still existed before the lane change was done and that part must be added).

3.1.1 Filtering log data

Throwing away log data with missing lane marker data
Many log data files had poor lane markings data, meaning that for some lane markings no measurements were registered for some of the lane markings sensors for a significant part of the recording. Having missing or incomplete data is not ideal in machine learning, which is why we decided to discard all log data files where the sensors did not register lane markings left and right of the ego vehicle for more than 99% of the samples. Essentially the two closest lane markings had to be present (detected) 99% of the time while the second lane markings to the left and right did not have to be present (detected) 99% of the time. The reparation and adding of lane markings was therefore done mainly to the second lane markings to the left and right.

Throwing away log data with no lane changes
We also decided to discard all log data containing no lane changes. The motivation for this decision was that we want the prototype trajectory generating model to be able to predict lane changing, instead of only predicting mere car following. We reasoned that the log data files containing lane changes would be to few in

percentage, in comparison to the files without lane changes, if we simply kept all log data files, which probably would impair the learning capabilities of the prototype trajectory generating model. The lane changes in the log data files were identified by checking if the vehicle had crossed a lane marking either to the right or to the left. For example, if the distance to the left lane marking went from low to high it means that the vehicle had passed the lane marking to the left and now detecting the next lane marking to the left as the left lane marking. Right lane changes were detected by the same, but opposite logic.

Throwing away log data with too low minimum velocity
Another filtering variable was the minimum velocity of the ego vehicle. We wanted to get rid of log data containing red lights, queues and highway take-off and exits, due to the worse quality of data in terms of identified lane markings. In addition, we wanted to get rid of log data of parkway driving and city driving. We therefore discarded log data files where the minimum speed of the ego vehicle was below 70 km/h.

3.1.2 Pre-processing

Reorganize lane markings data
The lane markings sensors of the ego vehicle registered nearby lane markings in terms of relative lateral position to the ego vehicle. The sensors were able to register a maximum of four simultaneous lane markings: two to the left of the ego vehicle, and two to the right. The implication of registering lane markings relative to the ego vehicle is that fields in the log data, representing the lane markings distances, keep track of different lane markings, depending on in which lane the ego vehicle is currently situated. To get rid of this dependency, all registered lane markings were given an ID and were re-ordered so that the most left lane marker during all samples in a log data file was given ID 0, and the next lane to the right was given ID 1, and so on. This re-organization of lane markers made it possible to find out the maximum number of lanes that were present in a log data file. This also made it possible to always send the same lane marker, defined by its ID, to the same input neuron.

Repair lane markings
After re-organizing the lane markings data we saw that some lane markings were missing, that is, set to zero for parts of the log data samples. We therefore wrote an algorithm to repair the lane markings by filling in the zero values, making use of the mean lane widths for each particular lane, and for each particular log data file.

Re-sample data
All log data was re-sampled to 4 Hz. This was done for performance reasons, since the training time of the model could be decreased to almost a tenth, compared to using all the samples.

Input
After re-sampling of the log data, the creation of the dataset could begin. The input-target pairs was created from each log data file as is described below.

The positions of the objects relative to the ego vehicle, the ego vehicle yaw-rate and longitudinal displacement since preceding sample, and the lateral distance to the lane markings were directly extracted from the log data file. We also wanted to add information on the future lane markings to the input, so we extracted lane markings measurements from future samples. From these future samples, a fixed number of p points, equally spaced longitudinally, was interpolated and converted to Cartesian Coordinates, with respect to the ego vehicle's position at the current sample. These interpolations were done for all p points and for all lane markings. The reason for adding coordinates for the future lane markings markers was to give the data driven model a fair chance of seeing the road, just as an ordinary driver can see the road ahead. The details of the interpolation algorithm can be seen in the project code.

After creating all input samples, the total mean and variance of each field was calculated. Each field could then be normalized to zero mean and unit variance. This is commonly done in machine learning to decrease the training time when training a data driven model. How the normalization was performed is shown in Equation 3.1.

$$x_{normalized} = \frac{x - \bar{x}}{s} \tag{3.1}$$

where $\bar{x}$ is the average of variable x over every sample and s is the standard deviation of every sample of x.

After normalizing the input samples, the samples were arranged in sequences of input samples. This was done because the data driven model is an RNN which takes a sequences of input features, before outputting an estimation of the target.

Target
The target of one input sequence was chosen to be a vector containing the future trajectory of the ego vehicle. The trajectory was represented in Cartesian Coordinates, with the ego vehicle's position in the last sample of the input sequence as origin. The target was calculated from the ego vehicle's longitudinal displacement and yaw-rate, and was then converted from Polar to Cartesian Coordinates.

3.1.3 Log data split

Dividing log data according to number of lane markings and lanes
While examining the remaining log data files we faced the problem of how to represent traffic scenarios where the number of lanes within each log data file was changing through time. Two very clear examples of these types of scenarios are when the ego vehicle is entering or exiting a highway. This could be identified in the log data as more than one lane appearing or disappearing after a lane change maneuver has

been performed. And sometimes, a highway exit may run parallel with the highway for a long distance, which gives rise to the problem of distinguishing between if a new lane has been created, or if the highway exit just runs parallel to the highway with a spacing of the same length as a highway lane.

The two fore-mentioned scenarios are hard to discriminate between, since the log data represent the different lanes in terms of registered lane markings relative to the ego vehicle's position. In addition, another problem of having many number of lanes when training a model on a particular dataset is that the number of fields in the input, designated for the lane markings feature, is not variable. Nor is it apparent how to represent non-registered lane markings. This reasoning lead us to the choice of making different data sets where log data files featuring roads with two lanes used in one dataset, and another dataset was made from log data featuring three lanes. No dataset containing one single lane was created, since no lane changes can appear in the one lane scenario.

Save input-target pairs to dataset
All input-target pair was then shuffled and divided into training and validation sets, together referred to as a dataset. This was done for both two and three lanes log data files, containing at least one lane change.

3.2 Model architecture

In this thesis, a system of models for vehicle trajectory predictions was developed. It was separated into two parts. The first part consisted of one or more predictive LSTM networks, which would serve as generators of prototype trajectory predictions. The second part was also an LSTM model, which had the job of assigning probabilities of which prototype trajectory was the most likely prediction. The first part of the system will in this chapter be referred to as the *prediction* or *prototype trajectory generating* model, while the second part will be referred to as the *decision* model. The outline of this chapter is as follows. Firstly, the architecture of the prototype trajectory generating model will be described. After that, the decision model architecture will be established. Continuing, the training procedure for the system of models will be explained, involving concepts such as loss function, hyperparameters, and weight initialization and optimization. Finally, different performance metrics for model evaluation are considered.

3.2.1 Prototype trajectory generating model

The prototype trajectory generating model consist of one or more connected LSTM layers and a fully connected output layer. The number of LSTM layers and number of hidden neurons in each layer is variable and are both considered as hyperparameters,

which can be fine tuned during the training process.

The input features to the first LSTM layer are:

- ego vehicle yaw-rate [Rad/s] with respect to (w.r.t.) the coordinate system of the ego vehicle's position in the previous sample;
- longitudinal displacement [m/sample] w.r.t. the coordinate system of the ego vehicle's position in the previous sample;
- estimations of lane markings (ID 0, ID 1, ID 2 and ID 3 if 3 lanes) [m] w.r.t. the coordinate system of the ego vehicle's position in the previous sample for some distance ahead in the form of coordinates for some number of points, based on lane markings estimations from future ego vehicle positions;
- surrounding object coordinates [m] w.r.t. the ego vehicle's position.

All input variables to the model are real numbers and the number of variables depends on the variables in the dataset being used. The model architecture is shown in Figure 3.2. The output layer takes the last LSTM layer's output signals as input, and transforms them into the same size as the target vector in the chosen dataset. This last step is needed since the output of the LSTM layers have the same size as the number of hidden neurons in each layer, which may not be the preferred size of the target. The output layer uses the linear activation function to get desired output values for regression.

3.2.2 Decision model

The decision model consists of one or more LSTM layers, just as in the prototype trajectory generating model. Inputs to the decision model are the same as to the predictive model. However, the fully connected output layer has the same size as the number of prototype trajectory generating models that are being used together in a system. The output signals of the output layer represents probabilities for each model on how likely their predictions are to be the most accurate prediction. The softmax activation function is used in the output layer to obtain these probabilities.

3.2.3 PyTorch

There are a few open source machine learning libraries available for the Python programming language. This thesis's machine learning components were implemented in the PyTorch library. PyTorch has mainly been developed by an artificial intelligence research group at Facebook. Two reasons why PyTorch was chosen is because it contains features such as tensor computation with efficient GPU acceleration and DNN which uses a tape-based autodiff system that enhances computation of gradients.

3.3 Model training

The model architecture depicted in Figure 3.2 describes the mode of operation for our system of models. First, the input data is fed to a model's decision network, consisting of some number of LSTM cells connected in series, where each cell have some number of hidden units in its hidden layers. The output of the LSTM block is then fed to a fully connected (FC) network with as many output units as there are predictive models in the system. Each FC output unit outputs the probability of its corresponding predictive model having the most accurate prediction as output. Thereafter, the predictive models are fed the same input data as was fed as input to the decision model. Each predictive model consists of some number of LSTM cells connected in series—each having some number of hidden units in its hidden layers—and a fully connected layer which maps the output of the LSTM block to the target dimension. The output of the whole system will therefore consist of many predictions, rated with probability of it having the highest accuracy.

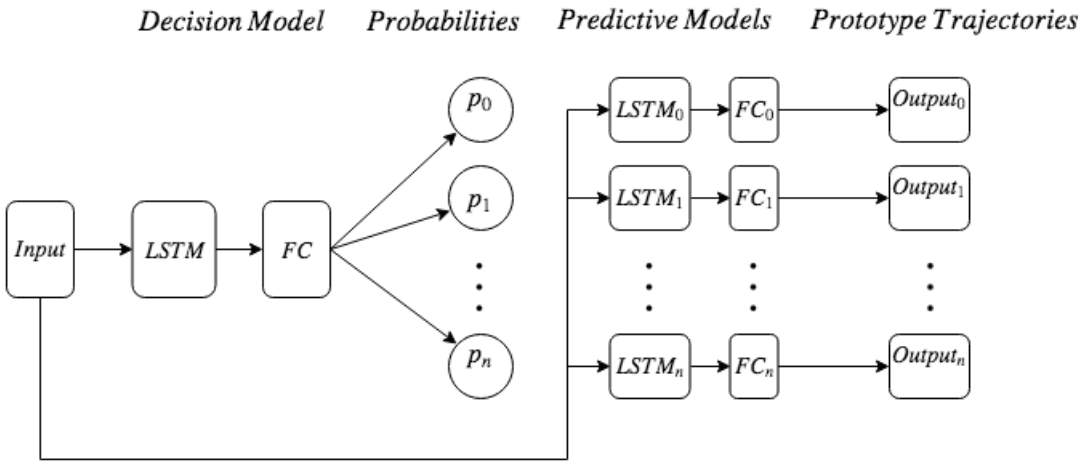


Figure 3.2: System architecture. The decision model determines which predictive model to use for a given input by calculating the probability to use each model. During training, each of the predictive models calculates their own output for the same input, then only the model which produces the most accurate output updates its weights. The decision model is then trained to predict which predictive model gave the best output. The hypothesis is that the different models will capture different characteristics within the data.

Training of the model is done in five stages. First, the training data is fed forward in each of the predictive models separately, yielding as many prototype trajectories as there are predictive models. Second, all the prototype trajectories are compared to the ground truth target trajectory. Third, the predictive model yielding the prototype trajectory with the lowest error, with respect to the target trajectory, will then proceed to be updated with backpropagation. Fourth, the input is fed forward into the decision model who's job it is to predict which of the predictive models to use in each traffic scenario. Fifth, the decision model will be updated by

backpropagation with respect to which predictive model gave the lowest prediction error.

The thought behind the predictive models is that they will capture different data and specialize on different decisions. The point is to produce predictive models that describes different driving behaviours where one might produce lane changes and another straight driving. Hypothetically, all the predictive models will give valid trajectories for each scenario given that the training is successful.

3.3.1 Loss Function

The loss function, L, defines what makes a trajectory prediction good or bad by comparing the output of the network with the correct answer. The loss function used for this work is mini-batch of the MSE of a predicted trajectory, which is defined as

$$L = \frac{1}{n_b} \sum_{j}^{n_b} \left(\frac{1}{n} \sum_{i}^{n} (y_i^{(j)} - \hat{y}_i^{(j)})^2 + (x_i^{(j)} - \hat{x}_i^{(j)})^2 \right) \tag{3.2}$$

where n_b is the batch size, n is the number of points predicted in the trajectory, (y_i, x_i) are the ground truth coordinates and ($\hat{y}_i$, $\hat{x}_i$) are the predicted coordinates.

3.3.2 Hyperparameters and grid search

A small hyperparameter grid search, described in section 2.1.6, for the number of layers and number of hidden units were conducted. The grid search showed that there were no distinct optimal numbers but rather that they could not be to small. We choose to have three LSTM cells connected in series and 100 hidden units in each layer. The learning rate varied slightly around 0.001 depending on how many prototype trajectory generating models were used and the number of lanes in the training dataset. Large batch size sped up the computing time but compromised the convergence of the network which lead to the use of a batch size of ten.

3.3.3 Weight initialization and optimization

As mentioned in section 2.2.5 weights are usually from some distribution and PyTorch default (which is we used) is $u(-\frac{1}{\sqrt{n_h}}, \frac{1}{\sqrt{n_h}})$ where n_h is the number of hidden units. With n = 100 gives us a uniform distribution between -$\frac{1}{10}$ and $\frac{1}{10}$.

The weights of the network are updated using backpropagation where the goal is to minimize the loss function in Equation 3.2 which forces ($\hat{y}_i$, $\hat{x}_i$) to approach (y_i, x_i) which makes the model produce predictions close to the ground truth. During backpropagation every weight in the network is updated. There are several gradient descent optimization algorithm that determine how to update the weights. We have chosen to use a popular gradient decent called Adaptive Moment Estimation (Adam). Adam uses the basic principles of gradient descent but is slightly altered to faster find a local minimum.

3.4 Model evaluation

How well a neural network performs (or any other algorithm for that matter) is dependent on the performance metric used. Generally the performance of ANNs are measured in mean error of some sort. But just because the mean is low doesn't tell everything about the performance of the model. Are there any outliers? How are the errors distributed? Is every error close to the mean which would make the average of all errors a good measurement or are the errors distributed far away form the mean on both sides? To better analyze the performance of the model the max, min and the distribution of the errors are measured.

3.4.1 Performance metric

The MSE will be used as performance metric, the error distributions of RMSE and classification of trajectories being in the left, current, or right lane, with respect to the road lane the vehicle currently is in.

3.4.2 Cross-validation

Cross-validation, described in section 2.1.7, was used to ensure that the model avoided overfitting to the training data. After looking at the cross-validation between the training and validation data, it was easy to use early stopping to ensure generalization.

3.4.3 Error distribution

Error distributions were used to understand how often the predicted trajectories were accurate, and how often they were completely off the target trajectory by looking at the mean and the maximum error in both longitudinal and lateral directions.

The error distributions show how far away the majority of predictions were from the target trajectory. They also show if there are any outliers with extremely high errors.

3.4.4 Confusion matrices: Truth versus actual

To understand if the predictions were accurate in terms of classifying lane changes confusion matrices was used to represent these relationships for each prototype trajectory.

4

Results

All findings from the thesis work are presented in this chapter. We begin by showing training and validation results for the trajectory prediction models and present the hyper parameters that were used. After that, an analysis of the longitudinal and lateral error distributions of the predicted trajectories are presented. Thereafter, we show a sensitivity and specificity analysis of the models by treating them as classifiers for changing lane to left, right or staying in the current lane. After analyzing the models, we benchmark our results to contemporary works.

At the end of this chapter, we display the outcome of a feature importance analysis made on the input variables of the different models, to show how much each variable contributes to the prediction errors. Finally, we display our findings concerning how the size of the dataset and the length of the input sequence affect the accuracy of the trajectory predictions.

4.1 Model Training

In this section, results are shown for the two cases where predictive models were designed and trained to generate either one or three prototype trajectories. These both cases are further divided into two different cases, where either a dataset with roads made up of two or three lanes are being used. Experiments with models that generate five and ten prototype trajectories are also presented in Table 4.2. Since a higher number of prototype trajectories give more cluttered diagrams, it was decided to only show training diagrams for the cases of one and three prototype trajectories.

4.1.1 One prototype trajectory

The training results for the model with one single prototype trajectory is shown in Figure 4.1, where a dataset containing two lanes has been used in Figure 4.1a, and a dataset containing three lanes have been used in Figure 4.1b. All hyperparameters

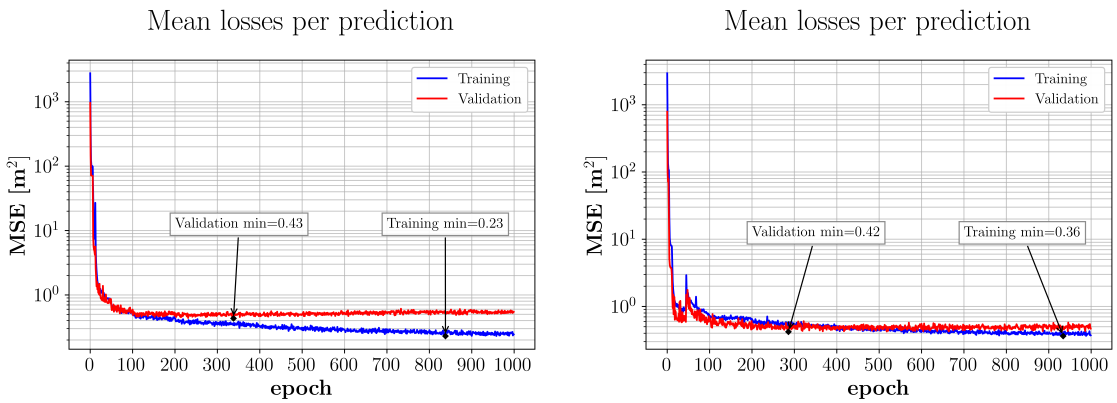
Table 4.1: Parameters used for training and producing results.

Parameters	**Values**
EPOCHS	1000
NBR_ITERATIONS_TRAINING	1000
NBR_ITERATIONS_VALIDATION	10
BACK_PROP_ALG	Adam
BATCH_SIZE_TRAINING	10
BATCH_SIZE_VALIDATION	1000
NBR_LANES	[2, 3]
MAX_DISTANCE	150
NBR_POINTS	4
NBR_LSTM_CELLS	3
HIDDEN_DIM	100
LR_START	0.001
LR_NEXT_ARRAY	[50, 100, 200, 400, 600, 800, 1000]
LR_UPDATE	0.5
Result metric	**Values**
MSE	-

and the result metric for the models in Figure 4.1 are shown in Table 4.1, and results for all experiments are presented in Table 4.2.

During training, many variations of hyper parameters were tried, and the ones which resulted in the lowest validation error are presented in Table 4.1. Some of hyper parameters we varied was the training batch size (BATCH_SIZE_TRAINING) in each iteration within an epoch; the number of hidden LSTM layers (NBR_LSTM_CELLS) inside the models; and the number of hidden neurons (HIDDEN_DIM) in each layer. We also tried many variations of learning rates (LR_START), where the rate was changed by an update factor (LR_UPDATE) at specific epochs given by the elements in an array called LR_NEXT_UPDATE. We also had some dataset parameters which affected the inputs to the models. The parameter called NBR_LANES describes if a two or three lanes dataset was used when training the models. The MAX_DISTANCE parameter decides for how far ahead of the ego vehicle the models get lane marker estimates, and the NBR_POINTS indicates how many road mark estimates that are used for each road marker. We also experimented with weight decay and dropout to decrease overfitting, but since the validation error did not decrease for any values on these parameters, it was decided to omit them from the parameter list. Manual experimenting is the basis for out findings, since extensive grid search on all parameters would grow exponentially for each sweep parameter. A final remark is that both the back propagation algorithm stochastic gradient descent and ADAM was tested, where Adam resulted in the lowest validation errors.

It can be seen in the training diagram in Figure 4.1a that the validation error (MSE) decreases with the training error until approximately epoch 350, where the MSE is equal to 0.43 $[m^2]$. Thereafter, the validation error saturates, while the training error



(a) Dataset with two lanes

(b) Dataset with three lanes

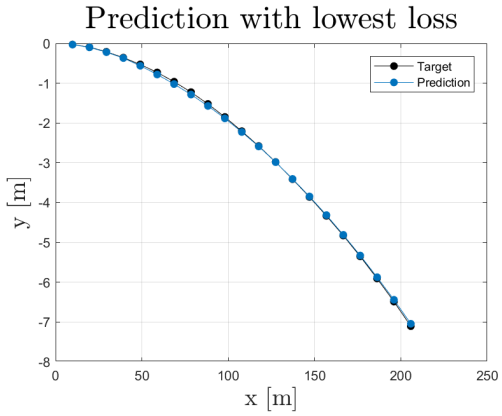
Figure 4.1: Training (blue) and validation (red) mean losses (per prediction) for one single LSTM model with (a) a two lanes dataset and (b) a three lanes dataset.

keeps decreasing and reaches its minimum at approximately 850 epochs where the MSE is equal to 0.23 $[m^2]$. Training the model after 350 epochs will therefore give rise to overfitting, since the model specializes on the training data and is not getting any better at making predictions on the validation data. This result indicates that 350 epochs where enough to train the model with the chosen training data and to potentially get higher accuracy in the model, more training examples, different hyper parameters, or another model architecture would be needed.

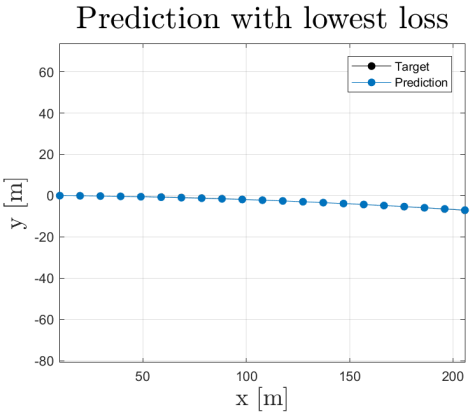
The training diagram in Figure 4.1b for the case with three lanes data shows that the validation error reaches its minimum at an MSE of 0.42 $[m^2]$ after approximately 300 epochs, and for the training data the minimum is found at an MSE of 0.36 $[m^2]$ after 950 epochs. Also in this figure, we see that more training would not improve the accuracy of the model for predictions on new examples, because of the same reasoning as for the training diagram in Figure 4.1a.

Comparing the training diagrams with the two versus three lane datasets in Figure 4.1 shows that the average prediction accuracy for the validation sets were almost the same, as well as the time it took to reach the minimum values. However, the error decreased much faster for the two lanes data case in Figure 4.1a, which probably is due to the fact that the driving options are more limited when there are two lanes instead of three.

Below in figure 4.2, figure 4.3, and figure 4.4, some predicted trajectories are compared to the target trajectories for predictions with the lowest, mean, and highest validation loss, respectively, where a three lanes dataset has been used. These figure are shown to give some illustration of how big the errors are from the smallest to the biggest errors.

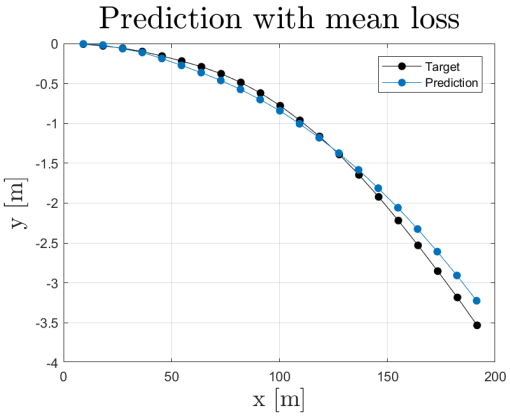


(a) Axis set to auto scaling.

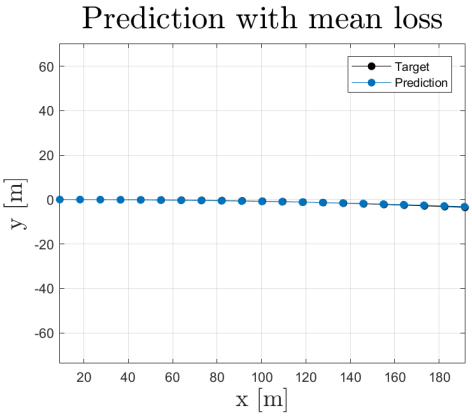


(b) Axis set to equal scaling.

Figure 4.2: Comparison between target and predicted trajectory for prediction in validation set with lowest loss. A dataset with three lanes where used.

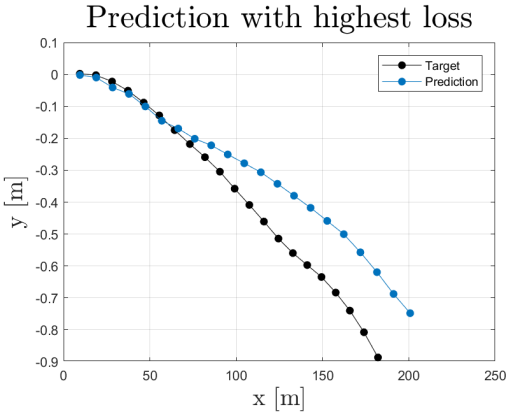


(a) Axis set to auto scaling.

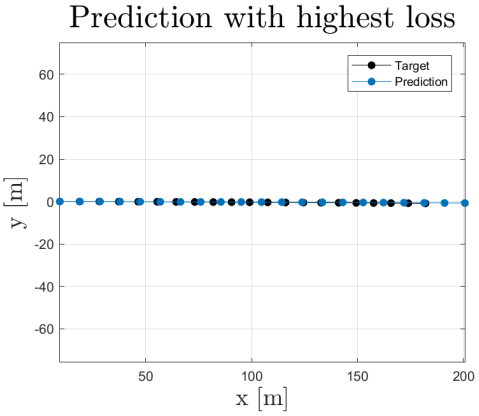


(b) Axis set to equal scaling.

Figure 4.3: Comparison between target and predicted trajectory for prediction in validation set with mean loss. A dataset with three lanes where used.



(a) Axis set to auto scaling.



(b) Axis set to equal scaling.

Figure 4.4: Comparison between target and predicted trajectory for prediction in validation set with highest loss.

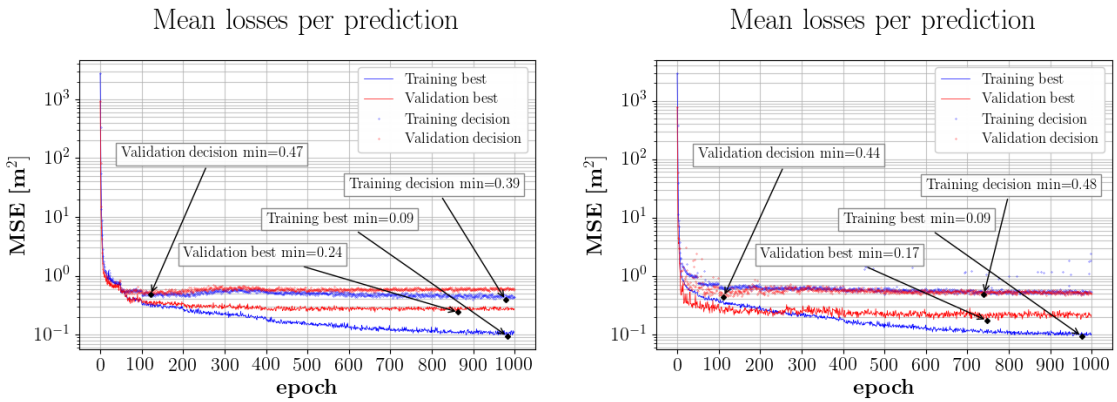
4.1.2 Three prototype trajectories

In the cases where we used more than one prototype trajectory, we had the same hyper parameters as in the single trajectory model, seen in Table 4.1. However, we measure the errors in a slightly different way than in the one trajectory case. First of all, there is one metric where we calculate the average MSE over all training examples for the current epoch, where we for each example selects the generated trajectory that yields the lowest MSE. This measure therefore shows how high the accuracy of the model would be if we for each input example would chose only the most accurate trajectory prediction. We will refer to this metric as the best, or sometimes lowest loss trajectory metric. since this metric does not give any help in deciding which trajectory is the best, we have another metric we refer to as the decided, or predicted (best) trajectory metric, where the decision networks inside our models have been trained of predicting which of the generated prototype trajectories is the most accurate one. This second metric will therefore be used when comparing the models with multiple prototype trajectories with the model with a single trajectory prediction.

In Figures 4.5 the best and decided trajectory metrics are shown for both training and validation losses for a model which generates three prototype trajectories. The dataset with two lanes have been used to train the model in Figure 4.5a and the three lanes dataset have been used in Figure 4.5b. For both the case of the two and three lanes datasets, the training best MSE keeps decreasing while the MSE of the corresponding validation MSE saturates already after around 400 epochs. The lowest MSE of $0.24[m^2]$ was found at around 860 epochs in the two lanes data case in Figure 4.5a, and after around 750 in the three lanes data case in Figure 4.5b, the lowest MSE was $0.17[m^2]$. This shows that models do not become non generalized in terms of the best trajectory metric, even if the models start to only fit their network weights to the training data in the last hundred epochs.

When it comes to the training and validation losses of the decision trajectory metric, the two and three lanes data cases differs somewhat in their results. The two lanes data model in Figure 4.5a has its lowest validation decision MSE of $0.47[m^2]$ at 120 epochs, and suffers from overfitting for the last hundred epochs. This overfitting does not seem to be resulting from overfitting the decision network in the model, since the training decision MSE follows the validation decision MSE. It is more probable that the prototype trajectory generating networks are becoming overfitted in regards to the validation decision metric, leading to the increase in MSE for the validation decision MSE. The three lanes data model in Figure 4.5b also saturates close to the same values, and the lowest MSE was found to be $0.44[m^2]$ around epoch 100.

The training and validation losses for each individual prototype trajectory generating model can be seen in Figure 4.6. The models are numbered zero to two to keep track of them, and only the validation losses are shown to keep the plots less cluttered. It is also shown in Figure 4.7 how often each model is chosen to have the lowest



(a) Data set with 2 lanes.

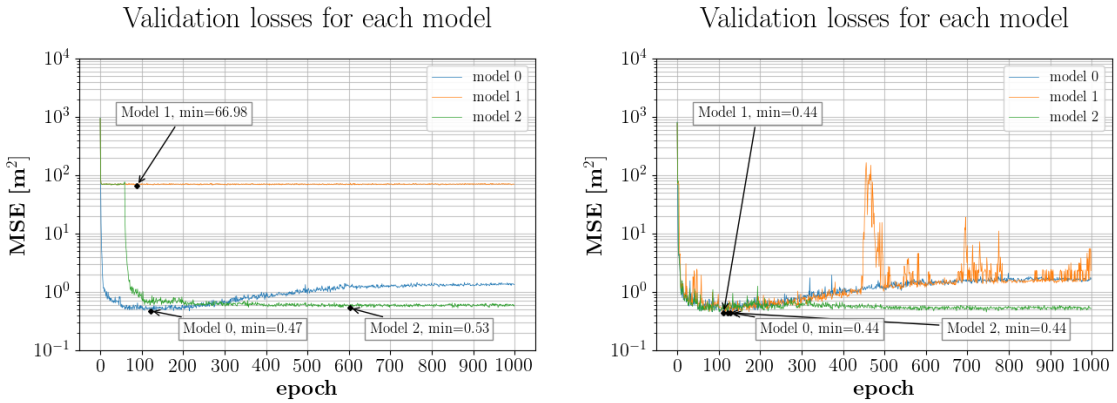
(b) Data set with 3 lanes.

Figure 4.5: Training (blue) and validation (red) losses for three jointly trained prototype trajectory generating LSTM models with (a) a two lanes dataset and (b) a three lanes dataset. Dots indicate losses where a decision model predicts which prototype trajectory generating network will be most probable to yield the lowest loss for each given input. The line plots indicate losses where, for each input, the prototype trajectory with the lowest loss was chosen.

MSE for its prototype trajectory, which results in more frequent updates of its network weights through backpropagation. If we start by considering the two lanes dataset number of updates plot in Figure 4.7a it shows that *Model 1* almost never has the lowest MSE, resulting in it not being updated very often. We also see that *Model 0* initially is chosen most often to generate prototype trajectories with the lowest MSE, but after around 300 epochs, *Model 2* starts to be chosen more frequently. By comparing Figure 4.7a to Figure 4.6 we see that *Model 0* in the first 300 epochs has the lowest MSE and has a minimum MSE of $0.47[m^2]$ at around 150 epochs, but increases its MSE in the following epochs. We also see that *Model 2* has the overall lowest MSE of $0.47[m^2]$ at around 600 epochs. By acknowledging that the validation decision MSE in Figures 4.5 is lowest after around 120 epochs, we have some evidence for the jointly trained model not improving after 120 epochs. If we think of the training dataset as two types of trajectory clusters. it seems as *Model 0* and *Model 2* specialized on one cluster each, to then exchange their initial cluster preferences at around 300 epochs. The third cluster of training data examples seems to be very small in number, and *Model 1* appears to become specialized on that cluster and has a minimum MSE of $66.98[m^2]$ at around 120 epochs.

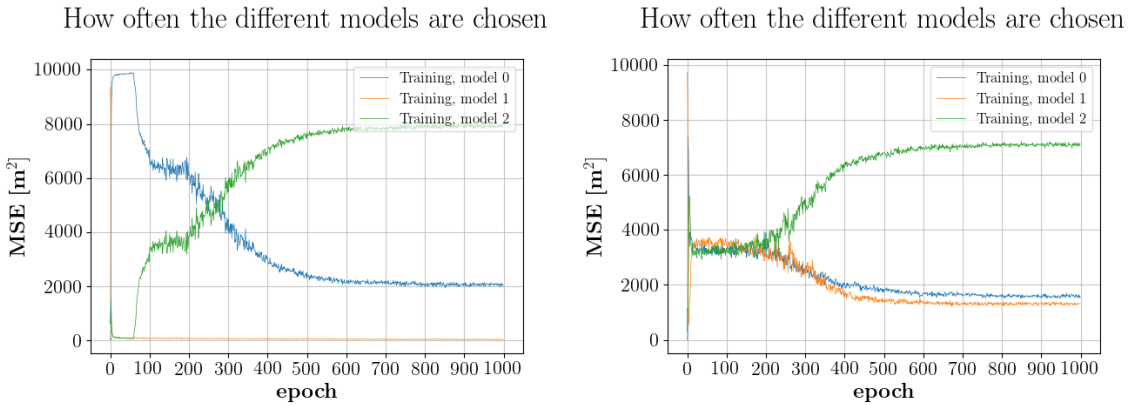
In the three lanes dataset in Figure 4.6b we see a different behaviour for the training convergence compared to the two lanes dataset in Figure 4.6a. This time, all models have the same lowest validation MSE of $0.44[m^2]$ after around 100 epochs and by looking at the validation decision minimum value in Figure 4.5b we can conclude that further training does not improve the validation decision metric. However, the validation best metric keeps decreasing to an MSE of $0.17[m^2]$ at around epoch 760. By looking at the update diagram in Figure 4.7b we now see that the three lanes training dataset's trajectories have been clustered in a different way compered to

the two lanes data. It appears as though *Model 2* specialized on a big cluster of trajectories and that *Model 0* and *Model 1* specialized on two smaller clusters.



(a) Dataset with two lanes. **(b)** Dataset with three lanes.

Figure 4.6: Validation loss during training.



(a) Dataset with 2 lanes. **(b)** Dataset with 3 lanes.

Figure 4.7: How often different networks are chosen during training.

4.1.3 Performance summary

In conclusion, we see that the the validation decision metric does not increase when we have more prototype trajectory generating models. However, the validation best metric increases for each extra added prototype trajectory generator. The MSE error for each model performed on both two and three lanes datasets can be seen in Table 4.2, for models with one, three, five and ten prototype trajectory generators. Here, **T** is to be interpreted as training, **V** as validation, **B** as using the best trajectory and **D** as using the trajectory chosen from the decision model. The **VB** column indicates that models trained on three lanes datasets give lower validation errors than the models trained on two lanes datasets. In contrast, the models'

performances on the training datasets (see the **TB** column) show the tendency of models generally having lower errors for two lanes than three lanes datasets. The same can be seen in the **TD** and **VD** columns, that the model performs better on two lanes training sets, but worse on the the validation sets.

Table 4.2: Five second Training (**T**) and validation (**V**) errors for models with **n** number of generated prototype trajectories, where a dataset with two or three **Lanes** have been used. The results show the MSE when choosing the lowest (**B** for best) MSE Prototype trajectory of each model, and when choosing prototype trajectories with aid of the decision (**D**) model to have the lowest MSE.

Model name	**n**	**Lanes**	**TB** $[m^2]$	**VB** $[m^2]$	**TD** $[m^2]$	**VD** $[m^2]$
LSTM_{1-2}	1	2	0.230	0.432	0.230	0.432
LSTM_{1-3}	1	3	0.363	0.420	0.363	0.420
LSTM_{3-2}	3	2	0.094	0.239	0.390	0.474
LSTM_{3-3}	3	3	0.093	0.173	0.480	0.437
LSTM_{5-2}	5	2	0.064	0.132	0.457	0.475
LSTM_{5-3}	5	3	0.067	0.110	0.563	0.443
LSTM_{10-2}	10	2	0.036	0.067	0.458	0.500
LSTM_{10-3}	10	3	0.037	0.051	0.553	0.423

4.2 Error distributions of trajectory predictions

In this section we analyze the error distributions of the one and ten prototype trajectory generating models, trained on the two lanes and three lanes datasets, respectively. This analysis is done separately for longitudinal and lateral errors to identify the error evolution in both the longitudinal and lateral driving directions. We here use the root mean square error (RMSE) to get the errors in units of meters instead of meters squared. This also enables for comparison of the lateral RMSE of the single prototype generating model with contemporary works, where the errors are shown at some future time steps.

4.2.1 One Prototype Trajectory

The validation errors at future predicted trajectory points: 0, 10 and 20, are shown as histograms in Figure 4.8, which map to the time values: $0.25[s]$, $2.75[s]$, and $5.25[s]$, respectively. Each successive predicted point has a time difference of $0.25[s]$. The three first diagrams in the upper row of Figure 4.8 display the longitudinal errors, and the three first diagrams in the lower row display the lateral errors. The fourth, most right, upper and lower diagrams in Figure 4.8 shows how the average error for each step. The upper plots show the longitudinal errors and the lower ones show the lateral errors. The longitudinal error seem to be growing exponentially,

but an odd thing we see for the lateral error growth at time step $t \approx 4$ in the lateral distribution diagram (to the lower right) is that the error is not strictly growing, as would be expected. The reason for this is unclear, and further investigation would be needed to understand this phenomenon.

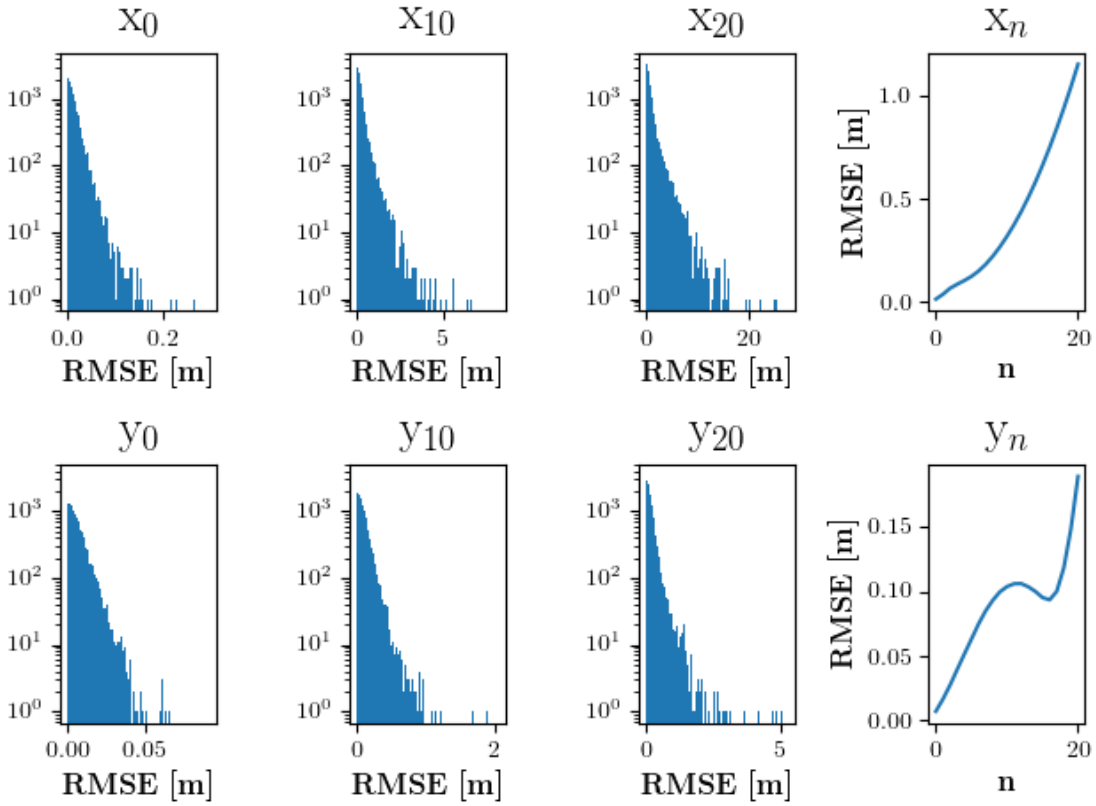


Figure 4.8: Longitudinal (x) and lateral (y) error distributions for a single prototype trajectory generating model, trained on a two lanes dataset.

The lateral error at time steps $t \in \{1, \ldots, 5\}$ are $0.04[m]$, $0.08[m]$, $0.11[m]$, $0.10[m]$ and $0.18[m]$, respectively, as can also be seen in the lower most right diagram in Figure 4.8. These values are also listed in Table 4.3, where the lateral error of the single prototype generating model, trained on the two lanes dataset, is compared to contemporary works. This table was taken from [1] where many different kinds of trajectory prediction models where compared. These models where all trained on the NGSIM [2] dataset, which is a dataset consisting of highway traffic data collected from stationary cameras on the highways in USA. Since our model was trained on log data, this comparison is not fair and cannot be used to draw any conclusions regarding the accuracy of our model's performance. It however serves as an example of how large errors can be expected if we had used the NGSIM dataset, and we at least see the combination of our model and the training data we used resulted in higher performance compared to if we had used the NGSIM dataset.

Table 4.3: Lateral RMSE positional errors in meters from related works. The table is taken from [1] and extended with data from our model ($\mathbf{LSTM}_{1-2}$). All models have been trained on the NGSIM [2] dataset, except $\mathbf{LSTM}_{1-2}$, which has been trained on the datset two lanes dataset used throughout this paper.

	Prediction horizon					
Model	1s	2s	3s	4s	5s	6s
$\mathbf{LSTM}_{1-2}$	0.04	0.08	0.11	0.10	0.13	-
Reference*	0.11	0.25	0.33	0.40	-	0.53
Type*	0.39	0.39	0.44	0.48	-	0.53
No f f*	0.14	0.24	0.33	0.41	-	0.54
No bypass	0.80	0.82	0.85	0.88	-	0.93
Bypass before	0.33	0.38	0.43	0.46	-	0.52
Lin. activ.	1.38	1.39	1.40	1.42	-	1.46
2 LSTMs	1.25	1.26	1.28	1.29	-	1.33
3 dense*	0.34	0.38	0.44	0.50	-	0.59
[14]	0.11	0.32	0.71	-	-	-
Bagged	0.17	0.25	0.33	0.40	-	0.46

4.2.2 Ten prototype trajectories

For the ten prototype trajectory generating model we have chosen to look at the validation best metrics error distributions in the case where the model was trained on the three lanes dataset. The validation errors at future predicted trajectory points: 0, 10 and 20, are shown as histograms in Figure 4.9, which map to the time values: $0.25[s]$, $2.75[s]$, and $5.25[s]$, respectively. Each successive predicted point has a time difference of $0.25[s]$. The three first diagrams in the upper row of Figure 4.9 display the longitudinal errors, and the three first diagrams in the lower row display the lateral errors. The fourth, most right, upper and lower diagrams in Figure 4.9 shows how the average errors changes for each step. The upper plots show the longitudinal errors and the lower ones show the lateral errors. We see here that neither the longitudinal nor lateral errors seem to be growing exponentially, which could perhaps be explained by the many prototype trajectories to choose from. Another observation is that the longitudinal and lateral errors for the 20th prediction point, x_20 and y_20, are equal to approximately $0.5[m]$ and $0.4[m]$, respectively, while the corresponding values in the single generating trajectory model in Figure 4.8 are approximately $1.1[m]$ and $0.15[m]$. It seems as the longitudinal predictions becomes much better when using more prototype trajectories, and that the lateral error grows when we use more lanes in our training data, which is to be expected, since more lanes lead to more driving options.

The ten prototype trajectory generating model will not be compared to any contemporary work, since it performed similarly to the one type trajectory generating model when looking at the validation decision metric (see Table 4.2), which is the

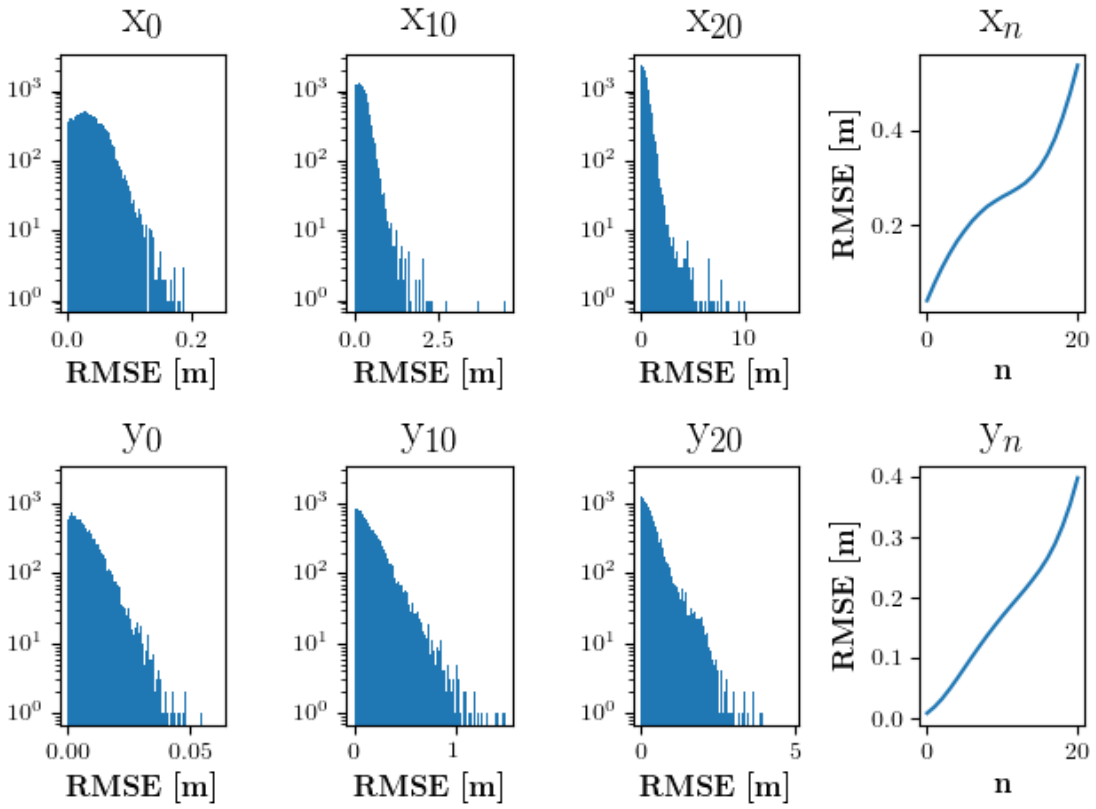


Figure 4.9: Longitudinal (x) and lateral (y) error distributions for a ten prototype trajectory generating model, trained on a three lanes dataset.

only metric that could be used when comparing the model to contemporary works. We will instead continue by analyzing the characteristics of the predicted trajectories in the next section.

4.3 Sensitivity Analysis

The hypothesis for the jointly trained models, with more than one prototype generating models, was that they where going to learn different characteristics. To investigate the characteristics of the different prototype trajectory generating models, we perform a sensitivity (true positive rate) analysis. The prototype trajectories are in this analysis treated as classifiers for the classes: left lane change, staying in lane, and right lane change. In Figure 4.10 we see three diagrams over all trajectories chosen for having the lowest validation best metric from a model with three jointly trained prototype trajectory generating models. For each generated prototype trajectory, the ones with the lowest validation best MSE is plotted in blue, and the average of all those trajectories are plotted in red. The models are numbered from zero to two, starting from the top. There are no significant differences in the lat-

eral directions of the predicted trajectories for *Model 0* and *Model 2*, while *Model 1* only had lowest validation best MSE for straight trajectories. When it comes to the longitudinal differences, we can see that all models average trajectories differed in length, although not so much. It is obvious that the different models were good at predicting certain kinds of trajectories, but it is not clear why they specialized on what they did. Ideally, we wanted to see some specialization such as one model predicting left changes while another predicted right lane changes. We will analyze if we can see any of these tendencies by means of confusion matrices.

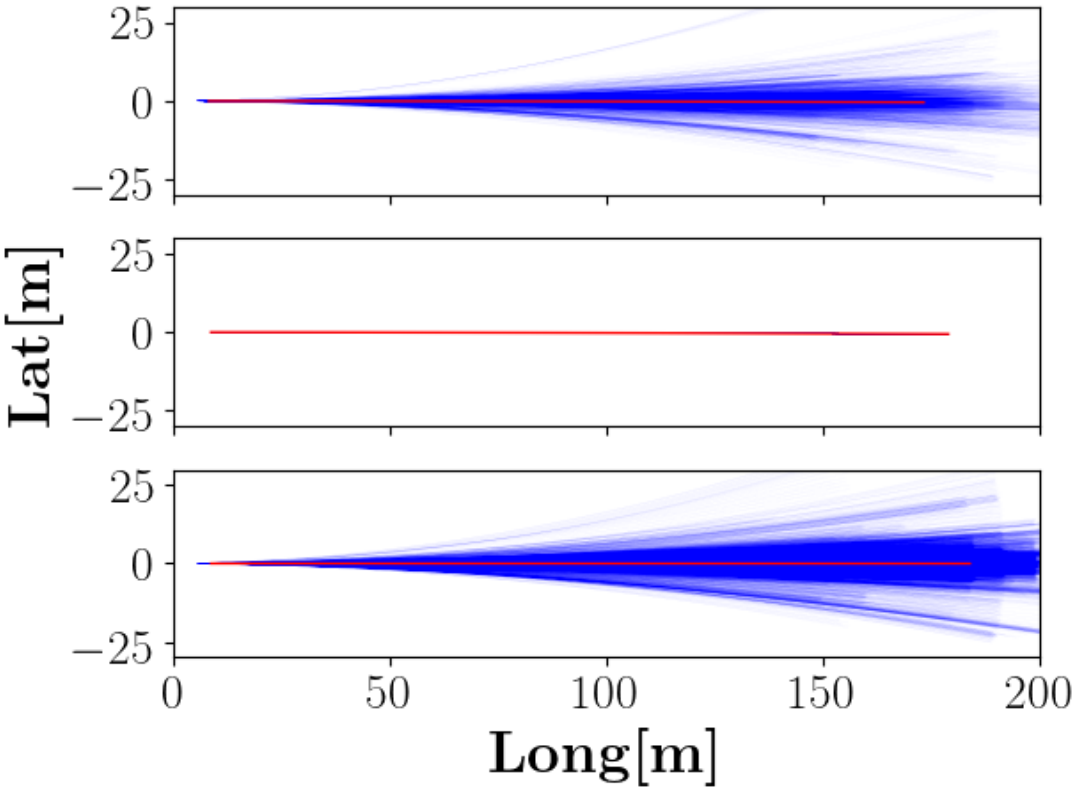


Figure 4.10: Trajectories for each prototype trajectory in a three prototype trajectory generating model, trained on a two lanes dataset. Only trajectories chosen for having the lowest validation best metric are plotted in blue, and the average of them is plotted in red.

In Table 4.4 we see the number of trajectories that was chosen for each prototype generating model in Figure 4.10. The trajectories are for each model in this figure divided into the three classes: lane change to the left, right or staying in the same lane, based on the ground truth trajectory for the predictions, and not what the models predicted. The models do not show any signs that they have specialized on any kind of lane change.

Table 4.4: Count of Left, straight and right ground truth lane changes for trajectory predictions for each prototype trajectory in a three prototype trajectory generating model, trained on a two lanes dataset. Only trajectories chosen for having the lowest validation best metric are counted.

Model	Left lane changes	Straight	Right lane changes
0	420 (8.2%)	4428 (86.7%)	262 (5.1%)
1	15 (7.9%)	148 (77.9%)	27 (14.2%)
2	746 (3.4%)	20132 (92.8%)	822 (3.8%)

The confusion matrices for all three prototype generating models' prediction are shown in Tables 4.5, 4.6 and 4.7, respectively, where only trajectories chosen for having the lowest validation best metric are counted. The percentages in the tables are the percentages of the sum of each row. The diagonal of the confusion matrices show the precision, that is, the number of times the predicted classification was actually right. All the confusion matrices shows that the models are much worse at predicting right lane changes than left lane changes. The predictions are compared to the actual values, and correct classifications are seen along the upper left to bottom right diagonal of the tables. We see that *Model 0* and *Model 2* accounted for almost all predictions and that they where best at predicting straight driving with a true positive rates of 99.4% and 99.7%, respectively. This is maybe not that surprising, since most of the time cars are driving in their current lane, and only a few trajectories involve lane changes, as can be seen by the trajectory counts in the tables. However, we also see a high classification accuracy for lane change prediction, where the left lane change true positive rates for *Model 0* and *Model 2* were 98.8% and 99.1%, and the right lane change true positive rates were 95.8% and 97.1%, respectively. The models where therefore slightly better at predicting left than right lane changes.

Table 4.5: Confusion matrix over predicted lane changes and actual lane changes for *Model 0*, trained on dataset with two lanes data.

Actual / **Predicted**	Left	Straight	Right
Left	415 (98.8%)	5 (1.2%)	0
Straight	11 (0.3%)	4402 (99.4%)	15 (0.3%)
Right	0	11 (4.2%)	251 (95.8%)

Table 4.6: Confusion matrix over predicted lane changes and actual lane changes for *Model 1*, trained on dataset with two lanes data.

Actual / **Predicted**	Left	Straight	Right
Left	10 (66.7%)	5 (33.3%)	0
Straight	0	135 (91.2%)	13 (8.8%)
Right	0	16 (59.3%)	11 (40.7%)

Table 4.7: Confusion matrix over predicted lane changes and actual lane changes for *Model 2*, trained on dataset with two lanes data.

Actual / **Predicted**	Left	Straight	Right
Left	739 (99.1%)	7 (0.1%)	0
Straight	38 (0.2%)	20067 (99.7%)	27 (.1%)
Right	0	24 (2.9%)	798 (97.1%)

In Figure 4.8 we see the confusion matrix for the single prototype trajectory generating model trained on three lanes data. This model predicted left, straight, and right lane changes with an accuracy of 99.3%, 96.5%, and 77.7%, respectively. This model was really accurate for predicting left lane changes, compared to the jointly trained model with three prototype trajectory generator models. However, it was worse at predicting straight driving, compared the the validation best metric. And it was especially inaccurate in right lane change predictions.

Table 4.8: Confusion matrix over predicted lane changes and actual lane changes for single prototype generating model, trained on dataset with three lanes data.

Actual / **Predicted**	Left	Straight	Right
Left	3204 (99.3%)	23 (0.7%)	0
Straight	682 (3.4%)	19584 (96.5%)	24 (0.1%)
Right	2 (0.1%)	774 (22.2%)	2707 (77.7%)

4.4 Variable importance

In this section we compare the importance between the input features being used in the training datasets, that is, how much each variable contributes to making accurate predictions. In Table 4.9 and Table 4.10 we see the variable importance (VI) for a single prototype generating model trained on two and three lanes data, after randomly permuting one or several variables at a time. Both tables reveal that

it is the speed of the vehicle that has the largest impact on the error of the model. The tables also show that the object tracks have the least effect on the models' validation error.

Table 4.9: Variable importance (VI) for a single prototype trajectory generating model trained on a two lanes dataset

Randomized feature	**MSE**
None	0.555
Objects	0.553
Yaw rate	0.575
Speed	163.848
Speed and yaw rate	164.508
First road line	2.357
Second road line	1.071
Third road line	2.468
All road lines	5.584

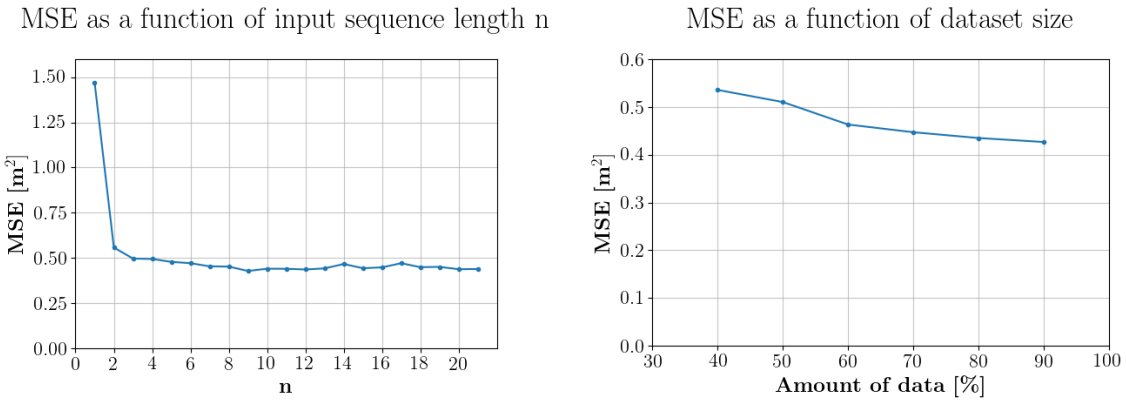
Table 4.10: Variable importance (VI) for a single prototype trajectory generating model trained on a three lanes dataset

Randomized feature	**MSE**
None	0.503
Objects	0.507
Yaw rate	0.515
Speed	232.742
Speed and yaw rate	238.200
First road line	2.731
Second road line	1.550
Third road line	1.886
Fourth road line	1.554
All road lines	7.023

4.5 Data size and input length dependency

In this section we present the results of an analysis on the size of the training datasets and the sequence length of the input that was fed to the models. In Figure 4.11a it can be seen how the validation error of the single prototype trajectory generating model, trained on a three lanes dataset, depends on the sequence length of the input. In theory, a maneuvering decision performed by a driver can depend on events that happened minutes or perhaps even hours before the current event. Figure 4.11a clearly shows that the driver model does not use information from more than a few seconds previous to the current state. There is a significant error decrease from

using one to using two time samples as input to the model. The difference in error is basically negligible after using more than two samples as input sequence. The model seem to be highly dependent on the information change between the current time step and the previous. The change in direction seem to play a huge factor in predicting a trajectory close to the target trajectory.



(a) How the mse varies with the input sequence length.

(b) MSE as a function of training data size.

Figure 4.11: MSE dependence on training data size (b) and input sequence length (a), for a single prototype generating model, trained on (a) a three lanes dataset and (b) a two lanes dataset.

In Figure 4.11b we see a direct correlation between lower validation error and larger dataset size used for training. However, it seems as though the decrease in validation error is slowly decaying, and we cannot for certain know that the model would have performed much better if we had used a larger dataset for training the model.

5

Discussion

In this chapter of the thesis, the results presented in **chapter 4** will be thoroughly discussed and analyzed. The figures and tables will be interpreted and explained. We will also reason about how the input data together with our models gave these results and how to potentially improve the results.

5.1 Training and validation losses (MSE) and performance summary

The summarized training results in table 4.2 showed that the average errors decreased with number of predictive models for the prototype trajectories having the lowest loss, while the errors for the prototype trajectories chosen by the decision model increased. These results indicate that the decision model was for the most parts not able to make any better predictions from all the prototype trajectories than a single predictive model would do. Consequently, this system of models can not be used to choose when some prototype trajectory are more suitable than others. However, the different trajectories can still be useful for getting insight into what different types of trajectories are most likely to arise from certain scenarios. The prototype trajectories could perhaps be sampled from when creating synthetic driver data that mimics the data the model was trained on, which could be useful for various simulations.

5.2 Error distributions

There are two interesting parts of Figures 4.8 and 4.9. The first is the shape of the average lateral error y_n in Figure 4.8. The curve has a local maximum and a local minimum. This is strange since it intuitively should be harder to predict the future position further into the future. Why this is the case is unclear but one reason could be due to lane changes somehow.

The second thing to notice is that the total MSE for the model with ten predictive models, shown in Table 4.2, is lower than when using a single predictive model while the lateral error increases. This can be seen when comparing y_n in Figures 4.8 and 4.9. The longitudinal error, x_n, decreases which makes it possible for the total MSE to decrease, but it is strange that the decrease in total MSE is so low when the lateral MSE increases.

5.3 Dependency on data size and length

Dependency on input sequence length shows that basically only two samples of historical data is sufficient. It does not care about history previous to that. It seems to be that the model looks for the moving pattern between the two points, to derive derivatives of the state of the ego vehicle. A hypothesis from this is that the sideways movement is the major information used to predict lateral movement such as lane changes. This would mean that the most significant information for the trained model is the change in movement from the previous point to the current point.

Looking at how the amount of data changes the MSE is interesting to see to find out if the amount of data used is sufficient or not. As Figure 4.11b shows, the MSE keeps decreasing with larger data size. At the same time it seems as though the gradient of the slope is also decreasing. Since we do not have more data we can not find out what would happen if we had more data, but the figure suggests that the MSE should decrease with data size. The question is how much it would decrease. The slope seems to (and should) stagnate at some point. If we are close to that point is hard to know. We want our data to contain enough scenarios to represent the scenario space. To see this we look at the slope and what happens if we add more data. A problem with this is that this is that it is dependent on the model. We already know that the model does not care about the other objects. Other objects are a major part of what defines a scenario. One reason why the slope is so flat could be because the model does not care about the other objects and therefore the scenarios look the same and adding more data will not make a significant difference.

5.4 Variable importance

As mentioned before, the speed is the most significant variable while the objects surrounding the ego vehicle have the lowest importance, which can be seen in Tables 4.9 and 4.10. The fact that randomly permuting the speed would have the largest impact is not surprising since the speed is the most important predictor when it comes to how far the vehicle will travel. The purpose of this thesis is to predict trajectories for different traffic scenes. Different combinations of cars' relative positions should

of course play an important role in what decision a driver makes and where possible future trajectories can take place, for instance, if a lane change is possible or not. The fact that surrounding objects to the ego car does not play any role for the future trajectory shows that the model is not truly making decisions on the entire traffic situation. This is equivalent to driving without any other nearby vehicles at any time which shows that the model is insufficient.

5.5 What the models learned

The first thing to check was if the models had specialized on lane changes. The trajectories for each model on the data that gave the best performance are shown in Figure 4.10. The trajectories and average trajectories should show the models tendencies. Each of the three models in the figure have average trajectories that point straight forward and trajectories pointing in every direction (except model 1 which nearly has no trajectories at all). A trajectory turning is usually not due to a lane change but due to the road turning. Also a left lane change can take place while the road turns right. The models could still have specialized on lane changes which Figure 4.10 will not show. What the figure would really show is if the models had specialized on left turning or right turning roads, which the models have not.

Instead the number of predicted lane changes where the models had the lowest MSE were counted in Table 4.4. This table would show if any of the models would be biased towards making lane changes to either side or any lane change at all. This is not the case for any of the models. One thing this table does not disclose is if the lane changes were correctly predicted.

Tables 4.5, 4.6 and 4.7 are able to tell how good the models are at predicting lane changes. Both model 0 and model 2 are fairly good at predicting lane changes while model 1 is significantly worse. That is likely because it was outperformed by the other models early in training which lead to it never giving the lowest MSE and therefore never being trained.

5.6 Future work

The next step in the development of the trajectory prediction model is to make a profound study of how to decide which of the trajectories from the output is the best. This thesis only covers a very simple decision network which tries to predict the model which would produce the most appropriate trajectory, but no further analysis have been put into this decision network. Future work within this is essential is the split model if going to be put in practical use its intended use case. Perhaps the most important issues that needs to be addressed in future work is the handling of

the objects surrounding the ego vehicle. An investigation regarding why the objects do not have an significant impact on the future trajectory predictions is needed.

6

Conclusion

A single LSTM model was trained on making vehicle trajectory predictions based on real world driving data at a minimum driving speed of 70 kph. It was found that the trained model predict future trajectories with an RMSE of $0.66[m]$ for a dataset containing two lane roads, and $0.63[m]$ for a dataset containing three lane roads, where a prediction is made up of 21 points covering five seconds of future driving. The maximum longitudinal and lateral RMSE for the last predicted point, five seconds into the future, were found to be $26[m]$ and $[5[m]$, respectively.

Multiple models were also trained jointly to generate multiple prototype trajectories, where the goal was to make each prototype trajectory specialize on different types of traffic scenarios and driving maneuvers. The prediction error between ground truth trajectories and the best prototype trajectory for each prediction, decreased significantly with the number of generated prototype trajectories (by a factor of 9 between 1 and 10 generated prototype trajectories). The different prototype trajectory generating models specialized on different scenarios, but the characteristics of the scenarios are not clearly separated. By our results it is nether left or right turns nor left or right lane changes.

Another model was also trained on deciding which prototype trajectory of the jointly trained models would yield the lowest prediction errors. Unfortunately, the prediction errors were larger compared to the single trajectory LSTM model.

None of the implemented models learned to utilize the surrounding object information in the training data in any useful manner and can therefore not be considered interaction aware, that is, they do not take the interactions with other road users or objects into consideration when making trajectory predictions.

Bibliography

[1] F. Altché and A. de La Fortelle. An lstm network for highway trajectory prediction. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 353–359, Oct 2017.

[2] U.S. Federal Highway Administration. Us highway 101 dataset, 2005.

[3] Adam Houenou, Philippe Bonnifait, Véronique Cherfaoui, and Wen Yao. Vehicle trajectory prediction based on motion model and maneuver recognition. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 4363–4369. IEEE, 2013.

[4] Stéphanie Lefèvre, Dizan Vasquez, and Christian Laugier. A survey on motion prediction and risk assessment for intelligent vehicles. *Robomech Journal*, 1(1):1, 2014.

[5] ByeoungDo Kim, Chang Mook Kang, Seung Hi Lee, Hyunmin Chae, Jaekyum Kim, Chung Choo Chung, and Jun Won Choi. Probabilistic vehicle trajectory prediction over occupancy grid map via recurrent neural network. *arXiv preprint arXiv:1704.07049*, 2017.

[6] World Health Organization et al. *Global status report on road safety 2018*. World Health Organization, 2018.

[7] Mattias Brannstrom, Erik Coelingh, and Jonas Sjoberg. Model-based threat assessment for avoiding arbitrary vehicle collisions. *IEEE Transactions on Intelligent Transportation Systems*, 11(3):658–669, 2010.

[8] Chiu-Feng Lin, A Galip Ulsoy, and David J LeBlanc. Vehicle dynamics and external disturbance estimation for vehicle path prediction. *IEEE Transactions on Control Systems Technology*, 8(3):508–518, 2000.

[9] Jihua Huang and Han-Shue Tan. Vehicle future trajectory prediction with a dgps/ins-based positioning system. In *2006 American Control Conference*, pages 6–pp. IEEE, 2006.

[10] Jrg Hillenbrand, Andreas M Spieker, and Kristian Kroschel. A multilevel collision mitigation approach—its situation assessment, decision making, and performance tradeoffs. *IEEE Transactions on intelligent transportation systems*, 7(4):528–540, 2006.

[11] Ronald Miller and Qingfeng Huang. An adaptive peer-to-peer collision warning system. In *Vehicular Technology Conference. IEEE 55th Vehicular Technology Conference. VTC Spring 2002 (Cat. No. 02CH37367)*, volume 1, pages 317–321. IEEE, 2002.

[12] Panagiotis Lytrivis, George Thomaidis, and Angelos Amditis. Cooperative path prediction in vehicular environments. In *2008 11th International IEEE Conference on Intelligent Transportation Systems*, pages 803–808. IEEE, 2008.

[13] Ashwin Carvalho, Yiqi Gao, Stéphanie Lefevre, and Francesco Borrelli. Stochastic predictive control of autonomous vehicles in uncertain environments. In *12th International Symposium on Advanced Vehicle Control*, pages 712–719, 2014.

[14] A. Eidehall and L. Petersson. Statistical threat assessment for general road scenes using monte carlo sampling. *IEEE Transactions on Intelligent Transportation Systems*, 9(1):137–147, March 2008.

[15] Christopher Tay, Kamel Mekhnacha, and Christian Laugier. Probabilistic vehicle motion modeling and risk estimation. *Handbook of Intelligent Vehicles*, pages 1479–1516, 2012.

[16] PyTorch. https://pytorch.org/.

[17] NVIDIA CUDA GPUs. https://developer.nvidia.com/cuda-gpus.

[18] Vladimir Naumovich Vapnik. An overview of statistical learning theory. *IEEE transactions on neural networks*, 10(5):988–999, 1999.

[19] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] Vladimir N Vapnik and A Ya Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. In *Measures of complexity*, pages 11–30. Springer, 2015.

[21] Sayan Mukherjee, Partha Niyogi, Tomaso Poggio, and Ryan Rifkin. Learning theory: stability is sufficient for generalization and necessary and sufficient for consistency of empirical risk minimization. *Advances in Computational Mathematics*, 25(1-3):161–193, 2006.

[22] Balázs Csanád Csáji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24:48, 2001.

[23] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[25] Li Deng, Dong Yu, et al. Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[26] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural

networks for image classification. In *Computer vision and pattern recognition (CVPR), 2012 IEEE conference on*, pages 3642–3649. IEEE, 2012.

[27] Jon Russell. Google's alphago ai wins three-match series against the world's best go player, May 2017.

[28] 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.

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