CHALMERS

Using Deep Neural Networks for Lane Change Identification

Master's thesis in Computer Science - Algorithms, Languages, and Logic

RYAN DAMARPUTRA WIDJAJA

Department of Mechanics and Maritime Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2019

MASTER'S THESIS IN COMPUTER SCIENCE - ALGORITHMS, LANGUAGES, AND LOGIC

Using Deep Neural Networks for Lane Change Identification

RYAN DAMARPUTRA WIDJAJA

Department of Mechanics and Maritime Sciences
Division of Vehicle Safety
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2019

Using Deep Neural Networks for Lane Change Identification
RYAN DAMARPUTRA WIDJAJA

Master's thesis 2019:93
Department of Mechanics and Maritime Sciences
Division of Vehicle Safety
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Chalmers Reproservice
Göteborg, Sweden 2019

Using Deep Neural Networks for Lane Change Identification
Master's thesis in Computer Science - Algorithms, Languages, and Logic
RYAN DAMARPUTRA WIDJAJA
Department of Mechanics and Maritime Sciences
Division of Vehicle Safety
Chalmers University of Technology

Abstract

Lane change maneuvers are commonly performed by drivers in highway driving situations. It starts with the driver planning and making the decision whether a lane change maneuver is necessary according to the situation. Once the driver decides to do the maneuver, he/she starts to prepare themselves. Next, he/she changes their lateral position until the vehicle crosses the line between lanes. Finally after crossing the lane marking, the driver adapts to his/her new positions by stabilizing the vehicle. When lane change maneuvers are done incorrectly, accidents may happened which can be fatal to those involved in the crash.

Advanced Driver Assistance Systems (ADAS) include several functions that rely on lane change detection e.g., lane departure warning (LDW) system and lane change assistance system. These functions can be used to help driver perform a safer lane change maneuver. Having a system which can accurately retrieve and recognize the driving characteristics of a lane change maneuver will be beneficial for the development of ADAS. Identifying lane changes in driving data can be done manually by annotations, but it costs a substantial amount of time and money in case of large driving databases. A cheaper solution would be to use machine learning algorithms as they excel in this type of problem. Several machine learning algorithms, specifically artificial neural networks, have been used in many different research applications, including lane change predictions.

This thesis work included several steps. Firstly, driving data was retrieved from UDRIVE, which contains naturalistic driving data collected from various European countries. The data was processed into segments containing lane changes and baseline driving. It served as an input for training and testing using a sliding time window approach. Three neural networks were constructed to identify lane changes. One served as the baseline model, and the other two were variations of the baseline model called modified Long Short Term Memory (LSTM) and stacked LSTM, respectively. Training and testing were conducted to these networks using the same configuration and dataset. During the training process, some of the parameters were adjusted according to their performance and some could not be adjusted. Parameters that cannot be adjusted are called hyperparameters and they usually relate to the structure of a neural network model. Both the modified LSTM and stacked LSTM were subjected to parameter tuning, which is a process of changing various trainable parameter in order to make the model perform optimally. After parameter tuning was done, the best model was further evaluated by using cross validation.

Results have shown that the stacked LSTM model has the best performance among the three models. It managed to reach F1 score of 0.7178 and able to identify 95% of the lane change data. However, the stacked LSTM model has performance problems in terms of training time in identifying the transition phases of lane change maneuver. Several factors which contributes to this performances issue are identified, such as the imbalanced training data, the variables selection, the structure of the network, the number of trainable parameters, the hyperparameter settings, and how the raw input data is processed. If these issues are resolved, it is expected that the stacked LSTM would have a higher performance in the range of 0.85 in F1 score.

Keywords: lane change, machine learning, neural network, lstm

Contents

Abstract i

Contents iii

1 Introduction 1
1.1 Scope . . . 3

2 Theory 3
2.1 Time-series Data . . . 3
2.2 Machine Learning . . . 3
2.3 Artificial Neural Network . . . 4
2.3.1 Feed-Forward Neural Network . . . 6
2.3.2 Recurrent Neural Network . . . 7
2.3.3 Long Short-Term Memory . . . 8
2.3.4 Activation Function . . . 8
2.3.5 Loss Function . . . 10
2.3.6 Back-Propagation . . . 10
2.3.7 Dropout . . . 10
2.3.8 Batch Normalization . . . 11
2.3.9 Optimization Algorithm . . . 12
2.4 Evaluation . . . 13
2.4.1 Overfitting and Underfitting . . . 13
2.4.2 Cross Validation . . . 13
2.4.3 Confusion Matrix . . . 13

3 Methods 15
3.1 Tools . . . 15
3.2 Data . . . 15
3.2.1 UDrive Project . . . 15
3.2.2 Data Extraction . . . 15
3.2.3 Data Processing . . . 16
3.2.4 Statistical Analysis . . . 17
3.3 Neural Network Model . . . 17
3.3.1 Model Construction . . . 17
3.3.2 Hyperparameter Tuning . . . 21
3.3.3 Model Training . . . 21

4 Results 21
4.0.1 Neural Network Performances . . . 22
4.0.2 Hyperparameter Tuning . . . 25

5 Discussion & Conclusion 26
5.1 Neural Network Models . . . 26
5.1.1 Baseline Model . . . 26
5.1.2 Modified LSTM Model . . . 27
5.1.3 Stacked LSTM Model . . . 27
5.2 General Performance and Limitations . . . 28
5.3 Conclusion . . . 29

6 Future Work 29

References 31

1 Introduction

Highways have become an integral part of our transportation system. They accommodate high-speed vehicular traffic by having separate roads for each direction with multiple lanes for each road, limited access as they can only be entered or exited at a certain locations, and offers few to no obstacles during a driving session. In most European countries, there is a high speed limit on highways, varying from 90 km/h to 140 km/h (European Road Safety Observatory, 2018). These factors make highways very attractive not only for long distance driving, but also for short distance driving as it allows driver to bypass city roads in order to get to the destination quicker.

According to the European Road Safety Observatory (European Road Safety Observatory, 2018), European countries have invested heavily in the construction of highways. In 1990, the total length of highways in EU-28 was estimated to be 42.207 kilometres. This was increased by 72.5% to approximately 75.820 kilometres by 2015. Because of these extensions, people were able to travel to a different cities easier than before, so they had more incentive to use highways. In Germany, the mileage of cars on highways has risen from 203 billion kilometres in 2000 to 244 billion kilometres in 2016 (Ahlswede, 2018) while in the Great Britain, 68.7 billion miles out of 327.1 billion miles were driven on highways in 2017 (Department for Transport, 2018). Moreover, data from 2017 suggested a 10% increase in highway traffic over the last decade (Department for Transport, 2018).

In highway driving, several maneuvers can be observed. These include following the lane, changing lanes to overtake vehicles, following a vehicle, and stopping at the shoulder due to an emergency. Based on the study by Li et al. (2015), three maneuvers were identified as the most common maneuver performed by the drivers: free driving (i.e., keeping a single lane without interference from other vehicles), following a lead vehicle, and performing a lane change maneuver.

A lane change maneuver in the highway driving context refers to the deliberate movement of a vehicle towards another lane. According to Chovan, Tijerina, Alexander, and Hendricks (1994), a lane change is a deliberate and substantial change in lateral position of a vehicle, while Beggiato and Krems (2013) defined that a lane change moment is when the center of vehicle crossed the line between two lanes. In Fitch et al. (2009), it is stated similarly that a lane change is a driving maneuver that moves a vehicle to another lane where both lanes have the same travel direction. In Griesbach (2019), it is mentioned that a lane change maneuver can be divided into four phases: planning, preparation, crossover, and adjustment. Firstly, the driver plans and decides to perform a lane change maneuver according to the situation. Secondly, if the driver decides to execute the maneuver, s/he may accelerate or decelerate relative to the vehicles in the target lane. Thirdly, the driver accelerates their lateral velocity substantially until the vehicle cross the line between two lanes. After the crossing, the driver decelerates their lateral velocity to straighten the car. Finally, the driver adapts to his/her new position by accelerating or decelerating the vehicle relative to the lead vehicle until a safe distance is reached.

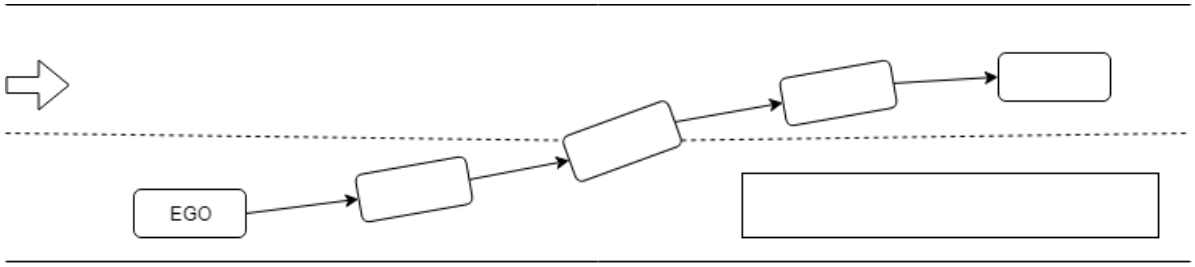

Figure 1.0.1: *A simple illustration of a lane change maneuver. The arrow indicates the driving direction, the smaller box represents maneuvering car, and the longer box represents a leading vehicle.*

Lane change maneuvers can be dangerous if it done improperly. Several law firms rated unsafe lane change as one of the top causes of road accidents Pines Salomon (2018), Adley Law Firm (2019), Lever & Ecker (2018), Ward Law Firm (2018). In Sen, Smith, Najm, et al. (2003), it is noted that in 1999, approximately 6.3 million crashes were recorded with 9% of them involving various scenarios of lane change maneuvers in the United States of America. 38.4% of these lane change crashes were categorized as a "typical lane change" scenario where one vehicle changing lane intentionally and colliding with another vehicle in the adjacent lane.

In European countries, Volvo Accident Research Team (2011) recorded 1.7 million traffic accidents on

average between 2005 and 2008. Approximately $55 - 65\%$ of them were crashes involving trucks causing car occupants injuries, and around 5% of these crashes were caused by a lane change maneuver in highways.

Advanced Driving Assistance Systems (ADAS) are systems that can be used to increase safety while driving a vehicle. Accurately identifying lane change maneuvers can be beneficial for the development of ADAS especially in several functions that rely on lane change detection e.g., lane centering, lane departure warning (LDW) system, and lane change assistance system. Drivers could be warned when they are about to change lanes in a way that is dangerous or violates traffic safety laws, thus reducing the possibility of crashing into another vehicle. In order to make an accurate identification possible, driving characteristics of a lane change maneuver must be retrieved and recognized by a potential system. Having an algorithm which can automatically recognize these characteristics can make the development process faster and more reliable.

Lane change maneuver data can also be retrieved and recognized manually by humans. This might seem easier to do as humans can be trained to recognize them in a short time. However, this approach has two problems. Firstly, it is not possible to do recognition in real time as humans can only examine what has happened in the past. Secondly, due to the sheer amount of driving data, it can be expensive in terms of time and money.

Machine learning algorithms are known for their ability to find the underlying pattern in a large amount of data and being used to make predictions based on what it has learned from the data. The accuracy of their prediction is determined by the quality of the training data. The higher data quality being put into a machine learning model, the better its prediction will be. Machine learning algorithms have been used extensively in several fields which provide a huge amount of data, such as image recognition (Simonyan & Zisserman, 2014), machine translation (Wu et al., 2016), traffic prediction (Lv, Duan, Kang, Li, & Wang, 2014), search engine suggestion (Ozertem, Chapelle, Donmez, & Velipasaoglu, 2012), and many more. Thus, applying machine learning algorithms seems reasonable also for a lane change identification.

Much research has been conducted on modelling driver's behavior using machine learning algorithms. In Tomar, Verma, and Tomar (2010) and Ding, Wang, Wang, and Baumann (2013), neural networks are used, which is a type of machine learning algorithm, to predict the lane change trajectory. In Olabiyi, Martinson, Chintalapudi, and Guo (2017) and Jain et al. (2016), a complex neural network model is utilized to anticipate driver actions. In Cui, Ke, and Wang (2018) a neural network is applied for traffic speed prediction. In Leonhardt and Wanielik (2018), two neural network models are used that run in parallel to predict lane changes to the left and to the right, respectively. In Griesbach (2019), two algorithms are implemented: a neural network and an echo state network. Both were used to predict lane changes in urban areas. These approaches focused on the prediction task where a model is trained to predict the maneuver based on the current data.

In Mandalia and Salvucci (2005), a different machine learning algorithm is applied, called Support Vector Machine (SVM), to detect lane change maneuvers. This is different from the previously mentioned approaches as the model identified lane change maneuvers by classifying the given data as "Lane Change" or "Lane Keep". For this task, they used several variables, such as acceleration, steering angle, lateral acceleration, and various lane position variables. Since lane change maneuvers do not have a fixed time length, they opted to use fixed-length time windows. These windows moved across the data stream and captured variable values that were inside them. Both overlapping and non-overlapping time windows were tested. They achieved the best result, 97.9% accuracy, with an overlapping window of 1.2 seconds. However to achieve this result, they exclusively used lane positions as their dataset. Consequently, the result was relying only on lane positions and thus, the relevance of the other variables for lane change maneuvers was undiscovered.

In Dang, Fürnkranz, Biedermann, and Hoepfl (2017), a neural network model is proposed to identify lane change maneuvers. They used this model to approach the problem in two ways: as a regression problem i.e., predicting the maneuver, and as a classification problem. The two approaches only differ in the produced output of the model. The classification model produced the probability of the data being labeled as "Lane Change" or "Lane Keep" while the regression model produced a "time-to-lane-change" value which is the time left until the driver performs a lane change maneuver. In order to make the results comparable, they converted their regression result into a classification label by looking whether the driver is actually performing a lane change maneuver after the predicted time has passed. Compared to Mandalia and Salvucci (2005), in Dang et al. (2017) more variables were used. They categorized the variables into three groups: driver monitoring (e.g. head and gaze direction), vehicle information (e.g. velocity, acceleration, steering wheel angle, and steering wheel moment), and environment information (e.g. lead car distance, relative distance to the middle of the lane, and relative angle between the lane and the vehicle's longitudinal axis). They also encapsulated these variables into time windows which is similar to what Mandalia and Salvucci (2005) did. They reported similar results from both approaches: the classification approach reached 86% accuracy while regression approach reached

87% accuracy. Both have the best result when they used 2.5 seconds as the length of time window. However, they are still worse than the SVM model by Mandalia and Salvucci (2005). This is expected since a lot more information was processed than the SVM model. More information generally means better performance, but it is also more likely to make mistake as the model has to consider every information in order to make a decision. Furthermore, the proposed model architecture was quite simple and may contribute to the weaker performance.

Based on these works, lane change identification can be approached as a regression problem and as a classification problem. Although they only differ in the output produced, it was decided to approach this as a classification problem because the goal is to correctly identify lane change maneuver, not to predict whether lane change maneuver will be performed in the future or not. The neural network model proposed by Dang et al. (2017) is chosen as the baseline model because they used their model to classify lane change maneuvers with a good result. Two neural network models are adapted from a baseline model. All the models are trained using a naturalistic driving database consisted of data from different European countries. A selection of variables were chosen based on what Dang et al. (2017) and (Mandalia & Salvucci, 2005) used and what is currently available. The overlapping time window method is used to encapsulate the variables and each window is given a label.

The rest of this report is structured as follows. In Chapter 2, I briefly review the theory of machine learning and artificial neural networks related to the proposed neural network. In Chapter 3, I explain the tools that were used for this project, methods of data extraction and analysis, and introduce three neural network models, two of which are modified from the baseline model. The performance of these three models is presented in Chapter 4. The evaluation, discussion, and conclusion of the approach and the models performances are detailed in Chapter 5. Finally, details about the future work are outlined in Chapter 6.

1.1 Scope

Since deep machine learning methods, particularly neural networks, are used in this thesis, several limitations were identified:

- It is particularly known that training time of a neural network model depends on several factors: the complexity of the model, the size of the dataset, and the availability of computational resources. More complex models often means longer training time. Thus, only a limited amount of neural networks is constructed in this project.
- UDRIVE consists of driving data from cars and trucks in various situations and environments across European countries. Even with only highway driving data, it is computationally costly to use the entire UDRIVE database. Thus, a smaller dataset consisting of randomly chosen parts of the UDRIVE database with highway driving data from cars is used to be the training and test dataset.

2 Theory

2.1 Time-series Data

Time-series data is a set of observations taken at a specific time t (Brockwell, Davis, & Calder, 2002). It has three characteristics (Kulkarni, 2017):

1. The newly arrived data is regarded as a new entry
2. Typically the data is ordered by time, and new data arrives in time order
3. Time is the primary axis

2.2 Machine Learning

The term "machine learning" was first introduced by Samuel (1959) when he created an algorithm which can learn to play a game of checkers when given only the rules of the game, a sense of direction, and a list of parameters which he thought have something to do with the game. He argued that providing a minute and

exact detail of a solution is a time consuming and costly approach, and this detailed programming effort might be eliminated by programming computers to learn from experience.

Russel and Norvig (2010, p. 693) supported this argument by providing three reasons. Firstly, the designers cannot anticipate all possible situations, or states, a program might end up in. For example, if someone designs a program to play chess, they have to consider every possible board state during each turn and put the states into the program. As the amount of possibilities is so enormous, it is not humanly possible for the designer to consider everything. Moreover if they can somehow put every state into the program, the computational cost is too high. This will result in a program that needs countless hours to calculate its next move. Secondly, the designers cannot anticipate changes over time, e.g., a stock-prediction program must adapt when stock market suddenly changes due to an unseen factor. Lastly, the human programmer might not know how to program the solution himself or herself, e.g. we can instantly read hand-written digits because our brains are trained to recognize and know their meaning since we were young. However, we may be unable to build a program that acts like a brain which can learn hand-written digits unless a learning algorithm is used.

According to Russel and Norvig (2010, p. 695), learning can be differentiated by the types of feedback received from the learning process. These are: supervised learning, unsupervised learning, and reinforcement learning.

In **unsupervised learning**, the algorithm receives no feedback. Instead, it will learn patterns in the given input data. The most common task in unsupervised learning is clustering, which is a task of grouping data into several clusters in a way that data within the same group are similar. For example, grouping types of stars based on their characteristics. Since there is no predefined categories of stars based on their attributes, the labels are not predetermined and therefore cannot be served as feedback. Thus, the astronomers have to do clustering.

In **reinforcement learning**, the algorithm will receive some feedback in the form of reward or punishment, and the instruction usually is to maximize the reward, or minimize the punishment. For example, suppose a robot is put inside a maze. The robot will be penalized for every move it makes, and it is given the goal to get out of the maze as quick as possible. At first, the robot will make a series of random movement. If then robot manages to get out of the maze, a reward is calculated and given to the robot. Afterwards, the robot will be put back into the same starting point as before. Based on the previous experience, the robot will try to improve its movement in order to improve its reward at the end. After it gets out from the maze again, a reward is calculated and the robot will be put back into the same position just like before. This process is repeated multiple times until the optimum path is found and the maximum reward is achieved.

In **supervised learning**, the algorithm will be given some examples of input-output pairs. Based on these examples, it will learn a function which maps from input to output. The feedback from this type of learning is an error measure of how far the output of the algorithm is from the ground truth value. The closer the output is to the ground truth value, the smaller will be the error. The algorithm will adjust its parameters based on this error. For example, a model is created to identify images of cat and dog. Several images of cat or dog (but not both) are given to the model for training. At first, the model will make a random guess, i.e., a cat image will be labeled as a dog image. As training goes on, the model learns the feature of cats and dogs and able to make a better guess. Once training is finished, a final test is conducted to judge the performance of the model. By this point, the model should be able to guess correctly when it is given an image of cat or dog. The difference between supervised learning and reinforcement learning is in the reinforcement learning, every decision can impact the future decision whereas in supervised learning, every prediction is made independently i.e, they do not influence the future prediction. Supervised learning methods have been used to solve many different, such as weather forecasting (Abrahamsen, Brastein, & Lie, 2018), image recognition (Simonyan & Zisserman, 2014), and stock market prediction (Guresen, Kayakutlu, & Daim, 2011). These examples are done using artificial neural network which is one of the most widely used learning algorithm in supervised learning.

2.3 Artificial Neural Network

Artificial Neural Network (ANN) is a computational model which takes inspiration from the biological neural networks that constitutes brain of humans and animals (Wahde, 2008, p. 151).

There are 3 main components of a physical (e.g., human) neuron cell: Dendrite, Axon, and Synaptic terminals. First, input signals will be received by Dendrites and their dendritic branches. These signals will be transmitted through the axon. When the signals reached Synaptic terminals, it will be propagated to another neuron cell connected to these terminals, starting the same process for the other neuron. A simple neuron is

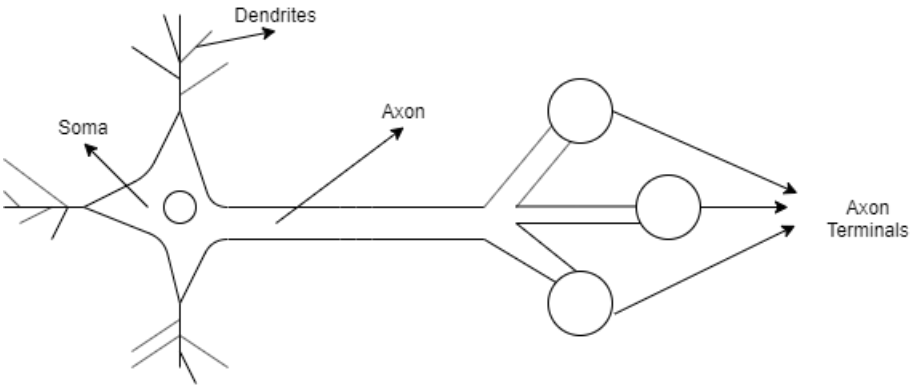

Figure 2.3.1: *A simple illustration of a neuron cell*

illustrated in Figure 2.3.1

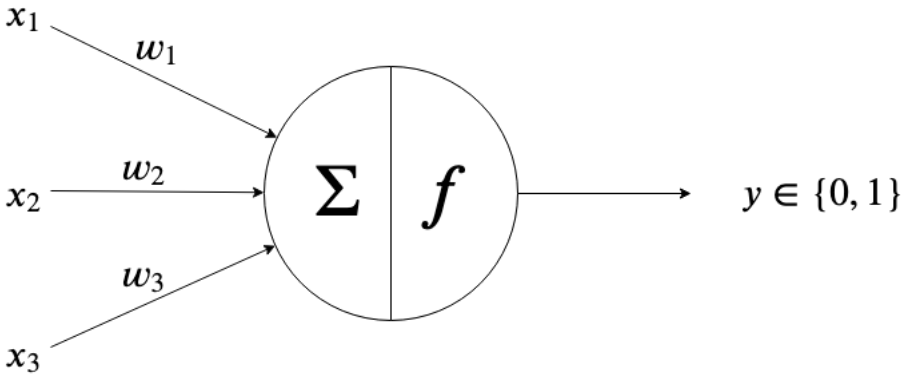

Figure 2.3.2: *An illustration of a McCulloch-Pitts Neuron.*

McCulloch and Pitts (1943) introduced a computational model based on this process, called **McCulloch-Pitts (MCP) Neuron**. In a MCP Neuron, a weighted sum of their inputs are calculated first. This can be mathematically expressed as:

$$o = \sum_{i=1}^{n} w_i x_i \tag{2.1}$$

where w_i are the connection weights, x_i are the input signals or the transmitted signals from another neuron, and o is the weighted sum.

Next, output signal y will be only propagated to the next neuron if the weighted sum exceeds a certain threshold θ. This can be mathematically expressed as below.

$$y = \begin{cases} 1, & \text{if } o \geq \theta \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

Threshold θ is one example of an **activation function** which responsible for activating a neuron.

MCP neuron model has two limitations. Firstly, it can only support boolean inputs, which means x_i can only support 0 and 1 as the value. Secondly, the weights are constant and only have two possible values: 1 and -1. Rojas (2013) noted that a MCP neuron is similar to conventional logic gates.

Fifteen years later, Rosenblatt (1958) proposed a more general version of a MCP Neuron known as **Perceptron**. It shares a similarity with the MCP Neuron, i.e., a Perceptron uses a threshold function similar to Equation 2.2 as the activation function. However, several changes were made.

Firstly, a constant **bias** is introduced. This bias serves as an offset to shift the input away from the origin (Hill, 2017). It is also giving neurons a chance to activate even in the absence of any input (Wahde, 2008).

Secondly, it now supports non-boolean inputs with weights associated to the inputs. These weights are now variable weights instead of constant weights like a MCP Neuron. Thus, Equation 2.1 is adjusted as follows:

$$o = \sum_{i=1}^{n} w_i x_i + b \tag{2.3}$$

where b is the bias term.

Secondly, a Perceptron uses **Hebbian Learning** which was first introduced by Hebb (1949). It was an attempt to understand the mechanism behind machine learning. He postulates that: "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased." This means the connection between two neuron cells will be strengthened if they fired simultaneously. In mathematical terms, the change of the connection strength denoted by w_{ij} over time can be described as

$$\frac{\mathrm{d}w_{ij}}{\mathrm{d}t} = \eta x_i x_j \tag{2.4}$$

where η is a constant that denotes the learning rate, and x_i and x_j denote the output of neuron i and j, respectively. A model of a Perceptron can be seen in Figure 2.3.3.

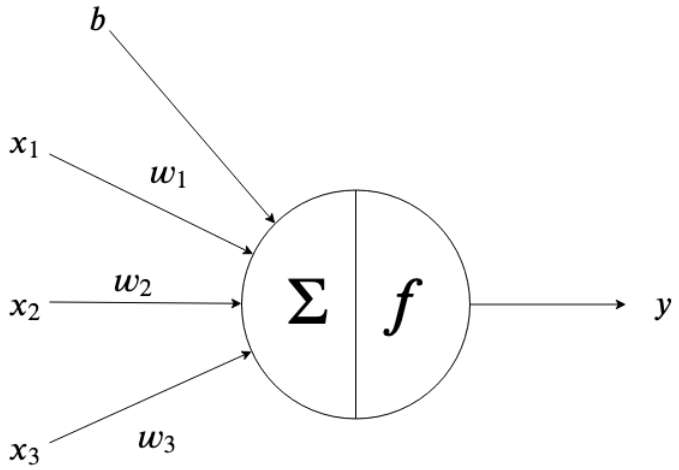

Figure 2.3.3: *A neuron in a perceptron model.*

This perceptron can be chained together to form an Artificial Neural Network. Typically, an ANN is arranged in the form of layers. There are at least three layers in an artificial neural network: the **Input layer** which is the input data itself, the **Output layer** which represents the desired output of one or several values, and the **Hidden layer** which is located between Input layer and Output layer. There can be multiple hidden layers in an ANN.

Parameters in an ANN can be separated into two types. The **trainable parameters** are parameters which can be adjusted as the ANN performs training, i.e., the connection weights between neurons and layers. The **hyperparameters** are parameters which cannot be adjusted during training process. These include the number of layers, the number of neurons in each layer, the learning rate value, number of epochs, dropout probability, and so on. Tuning the hyperparameters refers to the process of choosing their values so that the ANN achieved the optimal performance.

Based on how connections are made between neurons, there are two types of ANNs: **Feed-Forward Neural Network** and **Recurrent Neural Network**.

2.3.1 Feed-Forward Neural Network

A Feed-Forward Neural Network (FFNN) is a neural network which has connections only in one direction (Russel & Norvig, 2010). Thus, it forms a directed and acyclic graph as shown in Figure 2.3.4. Every neuron in a layer receives data from the previous layer, or directly from the input data. Then, a weighted sum of the inputs are calculated based on Equation 2.3. Next, an activation function will generate an output based on the

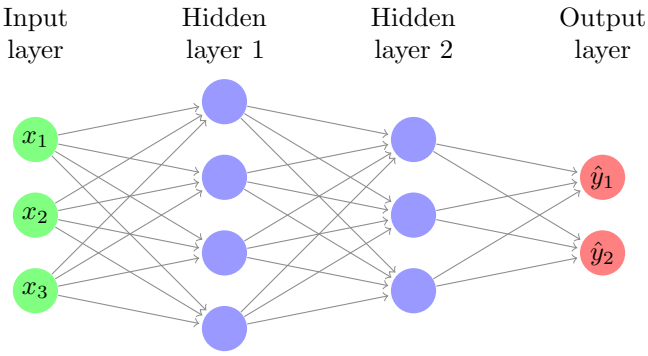

Figure 2.3.4: *An example of Feed-forward Neural Network with three inputs, two hidden layers, and two outputs.*

value of the sum, and finally the output value will be propagated forward to the next layer. There are no loops involved in this structure.

2.3.2 Recurrent Neural Network

A Recurrent Neural Network (RNN) is a type of neural network that delivers the output not only to the next layer, but also feeds it back to itself creating cycles. This may lead to a more chaotic behavior than FFNN since the respond of the current input depends on its initial state which may depends on the previous input. Hence, RNN can support short-term memory (Russel & Norvig, 2010).

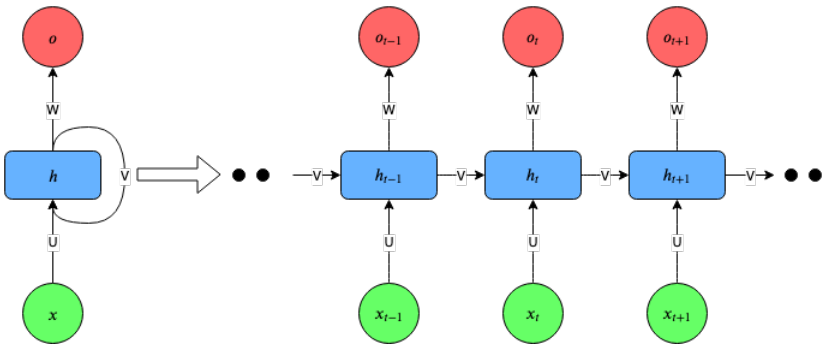

Figure 2.3.5: *An unfolded RNN figure*

As seen from the figure above, the RNN can use its internal state to process sequences of inputs and generate several outputs for every step. This means both the input and output can be variable in length.

Due to this property, their mathematical representation is adjusted:

$$\begin{aligned} h_t &= \sigma\left(Ux_t + Vh_{t-1} + b_h\right) \\ o_t &= \sigma\left(Wh_t + b_y\right) \end{aligned} \tag{2.5}$$

where h_t is the state of the hidden unit at time t, U is the connection weight between input x and hidden unit h, V is the weight between sequences, W is the weight between hidden unit and the output, o_t is the output of the RNN at time t, and σ denotes the logistic activation function.

Due to these characteristics, a RNN is suitable for a problem with a sequence of data as an input, e.g., a machine translation problem. However, Bengio, Simard, Frasconi, et al. (1994) discovered that it is difficult for RNN to learn long-term dependencies using traditional gradient descent for training because the gradient tend to vanish (approaching zero) or explode (approaching infinity). As a result, a RNN struggles to reach convergence in the training step because of the variations in the gradient and thus, a RNN will only capture the short-term dependencies instead.

To circumvent this problem, two approaches have been proposed. One of them is to use a better learning algorithm than stochastic gradient descent, e.g., gradient clipping approach by Pascanu, Mikolov, and Bengio (2013), using Hessian-free optimization approach by Martens and Sutskever (2011). Beggiato and Krems (2013) summarizes these methods as an alternative to a simple gradient descent for training a RNN.

The other approach is to augment the neuron itself and using a more sophisticated structure. The earliest example of this is a Long Short-Term Memory (LSTM) unit, proposed by Hochreiter and Schmidhuber (1997).

2.3.3 Long Short-Term Memory

Long Short-Term Memory (LSTM) was first introduced by Hochreiter and Schmidhuber (1997) as an answer to vanishing gradient problem. Figure 2.3.6 illustrates an LSTM cell along with the processes involved in the cell.

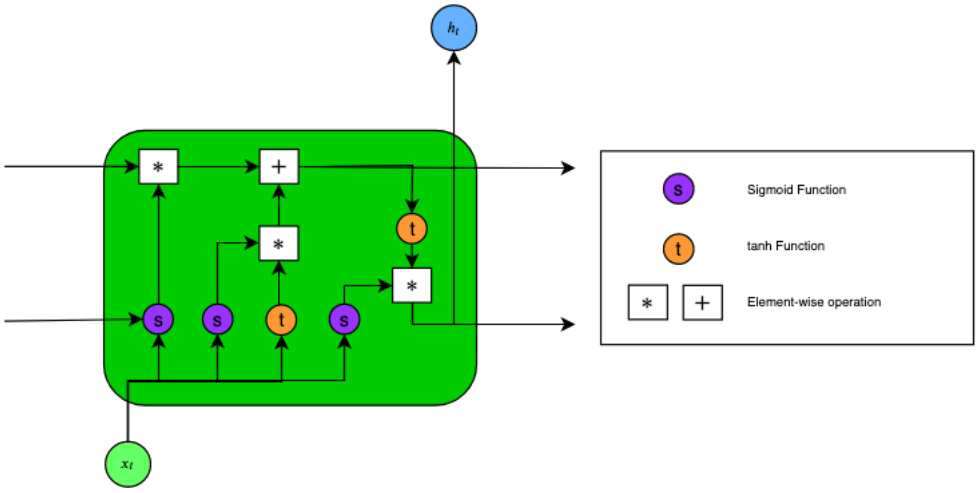

Figure 2.3.6: *An LSTM cell*

A LSTM cell consists of several gates: **input gate**, **output gate**, and **forget gate**. These three gates control the information flow within an LSTM cell. The Input gate decides which information will be updated, the output gate controls how much should be propagated from the cell, and the forget gate determines whether the previous state of the cell should impact the current state. This can be represented mathematically as:

$$\begin{aligned} f_t &= \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right) \\ i_t &= \sigma\left(W_i x_t + U_i h_{t-1} + b_i\right) \\ o_t &= \sigma\left(W_o x_t + U_o h_{t-1} + b_o\right) \\ c_t &= f_t \circ c_{t-1} + i_t \circ \tanh\left(W_c x_t + U_c h_{t-1} + b_c\right) \\ h_t &= o_t \circ \tanh\left(c_t\right) \end{aligned} \tag{2.6}$$

where W and U contains the weights of input and recurrent connections, σ and tanh denotes the logistic and hyperbolic tangent activation function, x_t, h_t, and c_t denote the input, output and cell state of the LSTM cell at time t. f, i, and o denote the forget gate, input gate, and output gate, respectively and lastly, every multiplication mentioned is an element-wise multiplication.

A cell state acts as an information carrier in a LSTM. It also serves as the "memory" of the network, i.e., it can carry information from earlier time steps to later time steps and this enables the network to "remember". As the process goes on, new information will be added or removed to the cell state with the help of the gates. So, cell states act as an information "highway" capable of carrying information for a long time. Consequently, the gradient will also flow backward using the same highway, thereby reducing the vanishing gradient effect.

2.3.4 Activation Function

In an ANN, the activation function computes an output value based on the weighted sum of inputs. This output value will determine if a neuron will fire or not (Nwankpa, Ijomah, Gachagan, & Marshall, 2018). One example of such a function can be seen in Equation 2.2.

Although they exist in every layer, activation functions have a different purpose depending on their position within the network. When it is located in the input layer or hidden layer, it serves to convert a linear mapping of the input to non-linear forms for propagation. However when placed in the output layer, it performs prediction instead.

There exist several activation functions that can be used in a neural network, but choosing a suitable one relies heavily on the type of problem at hand. For instance, a sigmoid function on the output layer might be suitable for regression or binary classification problems, but it is not suitable for a multi-class classification. Choosing an ill-suited activation function may also result in difficulties in the training process, i.e., slow convergence. Therefore, it is very common to use multiple types of activation function in an ANN. Below are the functions that was used in this project.

Sigmoid Function

The sigmoid function is a non-linear activation function commonly used in FFNN. It can be described mathematically as:

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.7}$$

The sigmoid function can appear in the hidden layer for keeping the output of a neuron bounded between $[0, 1]$. This will determine how much a neuron node can contribute to the next layer. The sigmoid function can also appear in the output layer for representing probability based outputs. For instance, in binary classification problems, for modelling logistic regression tasks and in other domains (Nwankpa et al., 2018).

There are also several drawbacks, such as slow convergence, gradient saturation, and non-zero centered output causing gradient updates to propagate in different directions which make optimization harder. Because of this, other activation functions were proposed to remedy these problems.

Hyperbolic Tangent Function

The hyperbolic tangent function, or tanh function, serves as an alternative to the sigmoid function. It has a zero-centered output and bounded in the range of $[-1, 1]$. This relationship can be described as follows.

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.8}$$

In a multi-layer neural network, tanh function gives a better training performance than the sigmoid function (Karlik & Olgac, 2011). The zero-centered output also prevents gradient updates to propagate in a different direction, thus helping in the learning process. However, due to this zero-centered property, the tanh function is also capable of producing a dead neuron, which is a neuron that will never contribute to any learning process because its activation function output is always zero.

The tanh functions have been used in the LSTM cell as seen from the previous chapter. It has also been used for natural language processing (Dauphin, Fan, Auli, & Grangier, 2017) and speech recognition (Maas, Hannun, & Ng, 2013).

Rectifier Linear Unit

Rectifier Linear Unit (ReLU) was first introduced by Nair and Hinton (2010). It performs a threshold operation where values less than zero are given zero as the output. This can be mathematically represented as:

$$f(x) = max(0, x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

From the equation, it can be seen that ReLU nearly retains the linear properties of the network since it does not compute exponentials or divisions. Thus, using ReLU with gradient-descent methods will make a faster convergence than using Sigmoid or tanh.

Softmax Function

The Softmax function is another alternative for an activation function. It computes a probability distribution from a vector of real numbers and produces a range of values between $[0, 1]$, with the sum of the values equal to 1 (Goodfellow, Bengio, & Courville, 2016, p. 181). The Softmax function can be described mathematically as:

$$f(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \text{for } i = 1, ..., J \tag{2.10}$$

The Softmax function is useful in multi-class classification problem. It will generate a set of probabilities with the target class having the highest. If the problem is a binary classification problem, the Softmax function will behave in the same way as the Sigmoid function. Thus, the Softmax function can be seen as a more general version of the Sigmoid function.

2.3.5 Loss Function

A loss function, in terms of artificial neural network, is a function that measures the difference between the ground truth value and value the generated by the network. Typically during the training step, the network will optimize itself by updating the connection weights based on the loss function and their values. The objective is to minimize the loss value. A low loss means the network is able to predict a value that is close to the ground truth. Thus, loss function dictates both the network's performance and how they should learn from errors.

Since the loss function is responsible for both performance and training, it must be chosen carefully as different loss functions have different purposes. For instance, the Mean Squared Error (MSE) function is widely used for linear regression problems, but it is not suitable for classification problems.

Cross-entropy Loss

Cross-entropy loss measures the performance of a neural network model that is used to solve a classification problem. Cross-entropy loss increases when the predicted probability value deviates from the ground truth value. A perfect model will have a cross-entropy value of 0.

Cross-entropy loss function can be described mathematically as:

$$\mathcal{L} = -\sum_{i=1}^{M} y_i \log \hat{y}_i \tag{2.11}$$

with M is the number of classes, y_i is the ground truth value, and $\hat{y}_i$ is the predicted value from the classification model. When cross-entropy is used to classify more than two classes, it is often called **categorical cross-entropy**.

2.3.6 Back-Propagation

When an input data is fed to a neural network, every neuron inside the input layer calculates the weighted sum of the input, passes the result to an activation function, and the final output will be transmitted to every neuron in the next layer. The process will be repeated for every hidden layer until it reaches the output layer. Once in the output layer, the network will produce a prediction value of $\hat{y}$ which can be compared to the ground truth value y to produce a loss value. This process is called **forward propagation** (Goodfellow et al., 2016).

Back-propagation, or **backprop**, is an algorithm that allows data to flow backwards from the output layer to the input layer using chain rule of calculus, carrying **gradients** from the loss function with respect to its parameters. These gradients will be used by another algorithm to optimize the strength of the connection weights between layers. A simple illustration of back-propagation can be seen in Figure 2.3.7.

2.3.7 Dropout

Neural networks can contain multiple hidden layers, making them able to learn complicated patterns and relationships between inputs and outputs and may improve prediction result. However, not all data is helpful during the training phase. A training dataset may contain noises and outliers that can interfere with predictions. A complex model can capture these noises and outliers as part of their learning. The model will try to cater to these noises and outliers if there are too many of them. Typically, this results in bad performance during testing phase. This is commonly known as **Overfitting**.

One of several methods to combat overfitting is the Dropout method. The term "dropout" refers to dropping out neurons in a neural network with a probability of p. The dropped neuron will not received nor transmit anything during training phase, temporarily removing them from the network. This will result in a simpler model which is beneficial to avoid overfitting. Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014) has shown that using dropout will result in a better generalization compared to other methods of avoiding overfitting. Figure 2.3.8 shows an example of a feed-forward neural network with dropout applied.

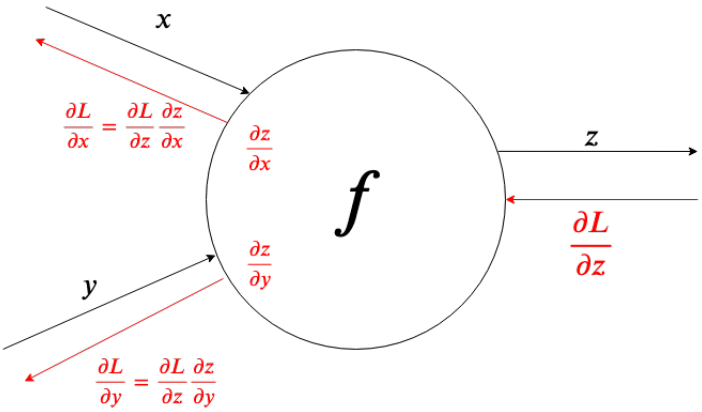

Figure 2.3.7: *The flow of information for forward propagation and back-propagation, highlighted in black and red respectively*

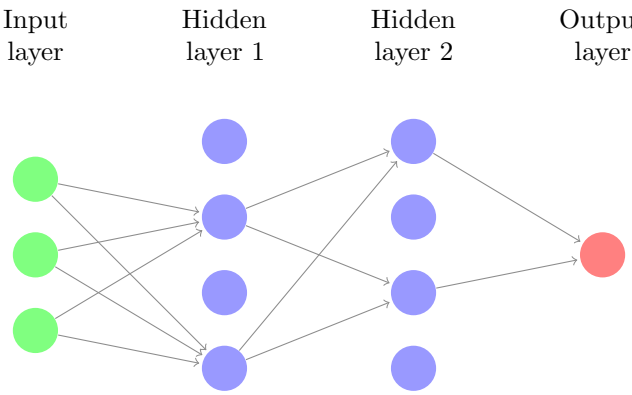

Figure 2.3.8: *An example of FFNN with Dropout applied, resulting in a simpler neural network model.*

Dropout can also be applied to a RNN and its variants. In this case, there are two types of dropout: regular dropout and recurrent dropout. The regular dropout works like the dropout in the feed-forward neural network, that is, it drops the neuron and temporarily disable them according to a certain probability. The recurrent dropout works slightly different. It drops the recurrent connections, that it, the connection between the time steps, according to a certain probability. This effectively makes the RNN ignoring the previous time step.

2.3.8 Batch Normalization

During the training phase, the distribution of each layer's inputs change as the parameters of the previous layer changes. This forces the learning rate parameter to be small and thus, slows down training time. Ioffe and Szegedy (2015) called this phenomenon as **internal covariate shift**. Internal covariate shift makes training hard because the layers need to adapt to the changing distribution of parameters, and a small change in parameters affects all preceding layers.

When training a neural network, giving input data in small batches can be beneficial (Ioffe & Szegedy, 2015). Firstly, the gradient of loss over a batch is an estimate of the gradient over the whole training data. The quality of the gradient estimate will be better as the size of the batch grows. Secondly, batch computation can be more efficient than individual computation due to the parallelism capability of modern computers.

Batch Normalization is an algorithm which speeds up training by performing normalization on a small batch of data after it has gone through the activation function. This reduces the internal covariate shift and utilizes the benefits from using mini-batch.

Batch Normalization takes values of x over a mini-batch $\mathcal{B}$ of size m as input. It introduces two learnable parameters γ, β to make sure that the transformation inserted in the network can represent the identity transform (Ioffe & Szegedy, 2015). Lastly, it will generate y_i as output which will transmitted to the next layer.

This algorithm consists of four equation which are executed in this order:

$$\begin{aligned} \mu_{\mathcal{B}} &= \frac{1}{m}\sum_{i=1}^{m} x_i \\ \sigma_{\mathcal{B}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i-\mu_{\mathcal{B}})^2} & \\ \hat{x_i} &= \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}^2} + \epsilon}} \\ y_i &= \gamma\hat{x_i} + \beta \end{aligned} \tag{2.12}$$

2.3.9 Optimization Algorithm

Optimization, in general terms, is the science of determining the best solution for a given problem, e.g., optimal models of manufacturing and management systems (Snyman, 2005). As concrete example: Given limited fuel in a truck, how should the driver press the gas and brake pedal in order to maximize the distance travelled? These problem can be solved using optimization algorithms. Likewise in terms of neural networks, how should one update the weights between each layer given a certain loss value? In this case, an iterative optimization algorithm such as **gradient descent** can be used to solve this problem.

Gradient descent is a first-order optimization algorithm that uses the **gradient vector**, or simply gradient ∇f, which is calculated by performing derivation on the loss function to determine the step direction. Once the step direction has been determined, it will be multiplied with the learning parameter η to determine the step size, and finally the parameters will be updated according to the step size. This process converges to the global or local optimum. Gradient descent can be described mathematically as:

$$\theta_{t+1} = \theta_t - \eta\nabla f \tag{2.13}$$

where θ_{t+1} indicates the updated parameter value and θ_t denotes the current parameter value.

In artificial neural network, the gradient vector is calculated based on the loss value. This gradient vector will be propagated backwards in order to update the connection weights between layers. The gradient vector calculation and propagation will be handled by the backprop algorithm, while the optimization algorithm, such as gradient descent, will determine how the update should be carried out. However, gradient descent has some problems (Walia, 2017). First, gradient descent can be very slow to achieve convergence because it will produce only one update after the gradient is calculated for the whole dataset. Second, it makes redundant computations for large datasets as it will recompute the gradient for similar examples (Ruder, 2016).

Several algorithms have been proposed to improve the performance of gradient descent. **Stochastic Gradient Descent (SGD)**, **Momentum**, and **Nesterov Accelerated Gradient (NAG)** focused on how to faster achieve convergence. Others such as **Adagrad**, **Adadelta**, and **RMpProp** were introduced adaptation of the updates to the individual parameters, i.e., perform larger or smaller updates depending on their importance (Walia, 2017; Ruder, 2016).

RMSprop

The idea behind Adagrad is to adapt the learning rate to the parameters: larger updates, i.e., high learning rates for parameters that are less important and smaller updates, i.e, low learning rates for parameters that are more important (Duchi, Hazan, & Singer, 2011; Ruder, 2016). This adaptation eliminates the need to manually tune the learning rate. However, Adagrad suffers from diminishing learning rates. This is making it harder to learn new knowledge as training goes on for long time.

RMSProp (Root Mean Square prop) was proposed by Tieleman and Hinton (2012) to tackle the diminishing learning rate with Adagrad. It involves using an exponentially weighted average of gradients. This weighted average will be used to calculate the new parameter. It can be represented mathematically as:

$$\begin{aligned} S_t &= \beta S_{t-1} + (1-\beta)g_t^2 \\ \theta_{t+1} &= \theta_t - \eta\frac{g_t}{\sqrt{S_t}} \end{aligned} \tag{2.14}$$

where both S_t and S_{t-1} are the weighted averages of the gradient at time t and $t-1$, respectively. The gradient is g_t at time t, η is the learning rate parameter, θ_t and θ_{t+1} are the parameters for time t and $t+1$, and finally, β is the constant parameter for the moving average. Tieleman and Hinton (2012) suggests β to be set to 0.9 and η set to 0.001. Chollet et al. (2015) states that RMSprop is a good choice for RNN.

2.4 Evaluation

2.4.1 Overfitting and Underfitting

Overfitting was mentioned in Section 2.3.7. To summarize, overfitting is a situation when a model is "memorizing" instead of "learning to generalize" from the dataset. This is due to the model capturing noises and outliers and adapting to them in the training data. An illustration about overfitting can be seen below.

Conversely, **underfitting** is a situation when a model is not able to capture the underlying patterns of the data. Underfitting will likely happened if the model has not enough parameters to represent the dataset. A comparison image of how underfitting, overfitting, and a robust model is shown in Figure 2.4.1.

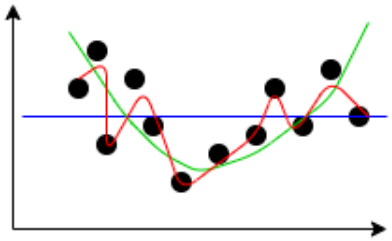

Figure 2.4.1: *The blue and red lines represent underfitting and overfitting respectively*

2.4.2 Cross Validation

Cross validation is a model validation technique which measures how well a model generalizes to an independent dataset (Drakos, 2018). In the most basic approach, which often is called **K-fold cross validation** (Pedregosa et al., 2011), the training dataset is split into k smaller sets. For each of these k folds, the following procedure is applied:

- The model will be trained using $k - 1$ folds as training data.
- The remaining data will be used as test data.

The performance measured by k-fold cross validation is the average of accuracy (or other types of measure). This method ensures every data is involved in both training and testing, resulting in a more general estimation of the model's performance. Furthermore, cross validation also prevent overfitting because the model will not be biased towards a certain part of the data.

2.4.3 Confusion Matrix

Confusion Matrix is a matrix used to evaluate the performance of a classification model. It has two dimensions: one dimension represents the predicted label, and the other represents the true label. The diagonal elements represent the number of correct predictions for which the predicted label is equal to the true label. Conversely, the off-diagonal elements represent the number of incorrect predictions made by the model. The higher the values in the diagonal elements the better (Pedregosa et al., 2011).

There are four variables involved when using a confusion matrix:

1. **True Positive (TP)**. This value represents the correctly predicted positive class, which means the actual class is positive and the predicted class is also positive.
2. **False Positive (FP)**. This value represents an incorrect prediction which occurs when the actual class is negative, but the model outputs a prediction in positive class.
3. **True Negative (TN)**. This value represents a correctly predicted negative class, which means the actual class is negative and the predicted class is also negative.
4. **False Negative(FN)**. This value represents an incorrect prediction similar to False Positive, but in the opposite direction. It happens when the actual class is positive, but the model outputs a prediction in negative class.

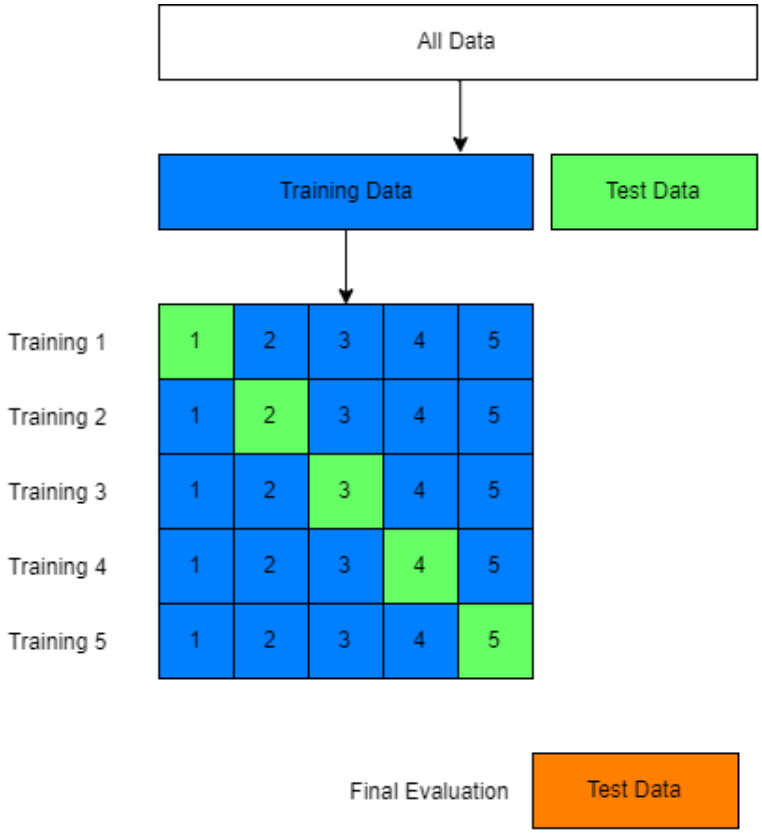

Figure 2.4.2: *How k-fold cross validation works with k=5*

		Predicted Class				
		1	2	3	4	5
Actual Class	1	TN	TN	FP	TN	TN
	2	TN	TN	FP	TN	TN
	3	FN	FN	TP	FN	FN
	4	TN	TN	FP	TN	TN
	5	TN	TN	FP	TN	TN

Figure 2.4.3: *An illustration of a confusion matrix with a multiclass classification problem.*

With these four parameters, several metrics can be calculated. One of them is **Accuracy** metric, which is calculated by dividing the TP value with the total observations. This metric can represent the general performance of a model if the dataset is balanced, i.e., each class has the same amount of data.

However, in real-life applications, most of the datasets are not balanced. For instance, suppose a model is built to identify driver's behaviour in an urban traffic. The behaviour is classified into three labels: "Following the road", "Left Turn", and "Right Turn", assuming the driver never puts the car in reverse gear. The data will most likely dominated by the data from "Following the road" label because in real life, the driver will spend most of his, or her, time just driving straight and following the road and making left or right turn as needed like in an intersection. He, or she, will not do left turn and right turn consistently all the time unless he, or she, is drunk. Therefore when this data is fed into the model and tested, there is a possibility that it can blindly put every data into "Following the road" label and still get a high accuracy. In this case, a high accuracy does not represent the general performance of the model. Thus, another metric is required to judge the true performance of the model.

F1 Score is an alternative metric to accuracy. It considers both **precision** (When the model predicts positive, how often is it correct.) and **recall** (When it is actually positive, how often does it predict positive.).

Their relationship can be described as follows:

$$\begin{aligned} precision &= \frac{TP}{TP + FP} \\ recall &= \frac{TP}{TP + FN} \\ F1 &= 2 \times \frac{precision \times recall}{precision + recall} \end{aligned} \tag{2.15}$$

As it can be seen in Equation 2.15, F1 score takes both FP and FN into account and it seeks balance between precision and recall (Shung, 2018; Joshi, 2016). Good F1 score is achieved when a model has low FP and low FN, which means the model is able to identify and classify data more accurate. A model is considered to be perfect if it has an F1 score of 1. Conversely, 0 F1 score means a model is a total failure.

In a multi-class classification problem, F1 score can be computed for each class. The final F1 score is calculated by averaging across multiple F1 scores (Pedregosa et al., 2011). There are multiple ways to calculate average F1 scores. When using macro average, F1 scores is calculated independently for each class and then take the arithmetic mean to get the final F1 score. This way, every class is regarded as equals. However when using micro average, F1 score is calculated once by applying precision and recall globally. This means the sum of TP, FP, and FN from every class are used. Thus, every sample is regarded as equals. This is the preferable method if the dataset suffers from an imbalanced class. Both these methods are useful to measure the performance of a multi-class classification model as it forces the model to have a good performance on every class if it wants to have a high F1 score in both micro and macro average.

3 Methods

3.1 Tools

Python (Van Rossum & Drake Jr, 1995) was chosen as the programming language for this project along with various packages e.g., NumPy (Oliphant, 2006) and Scikit-learn (Pedregosa et al., 2011) for mathematical computations, Matplotlib (Hunter, 2007) for plotting various graphs, and Keras (Chollet et al., 2015) for building neural network. Jupyter Notebook (Kluyver et al., 2016) was also used to organize code blocks. These packages were chosen for their simplicity and versatility.

3.2 Data

3.2.1 UDrive Project

The UDrive project was a large scale naturalistic driving study in Europe on cars, trucks, and powered two-wheelers (Barnard, Utesch, van Nes, Eenink, & Baumann, 2016). The aim of UDrive was to achieve a better understanding of driver behaviour (Bärgman et al., 2017). This is attained by conducting several tasks of which one is building a central database with the collected Naturalistic Driving Data (NDD).

The UDrive NDD contains eight video streams, data from the vehicle's Controlled Area Network (CAN), accelerometer and angular rate sensors, GPS, and a smart camera. Vehicle data was collected in various European countries. NDD from cars was collected in France, Germany, Great Britain, Poland, and the Netherlands, NDD from trucks was collected from the Netherlands, and NDD from powered two-wheelers (scooters) was collected from Spain. A tool called SALSA is used in UDrive to perform various tasks, e.g., develop, share, and apply algorithms for signal processing, inspect driving data in various situations, etc.

In this thesis, only the highway driving data from cars was used to create a dataset for the training, validation, and testing. The reasoning is that a lane change maneuver is easier to observe in highways compared to urban roads.

3.2.2 Data Extraction

In several studies involving lane changes, different researchers used similar variables to each other. Mandalia and Salvucci (2005) used acceleration, lane position, lateral acceleration, and steering angle. Meanwhile, Dang

et al. (2017) used speed, distance to next vehicle, head and gaze direction, and lateral distance together with the previously mentioned variables.

Based on this observation, nine signals, which have a frequency of 10Hz were extracted from the UDrive database. Eight of them are the combination of what Mandalia and Salvucci (2005) and Dang et al. (2017) were using except of the head and gaze direction. The last signal, E_LaneChange, is used as the basis of the labeling process. This signal detects lane change based on consecutive peaks in the lateral acceleration. The peaks are the first or the last maximum values higher than the 95^{th} percentile of the lateral acceleration within the same segment. The lane change start and end is defined as two peaks with a positive peak indicating the start of a lane change, and the negative peak indicating its ending. Between and including the start and the end, a value of one will be emitted by the signal. This indicates the driver initiating, performing, and ending a lane change maneuver.

Data extraction was done using the SALSA tool and MATLAB scripts. First, driving data was divided into two groups based on the road type: Highway driving, which consists of any driving data on urban and country motorways, and non-highway driving, which consists of driving on rural, urban (city), and slip roads. Then, segments were generated for the highway driving group where each segment contains the timestamps and the nine signals mentioned before. Finally, each segment was saved as a MATLAB struct and stored in a separate file. These files were then processed by Python, which will be explained in the next section.

3.2.3 Data Processing

Multiple steps were taken for processing the segments from the UDrive database. First, driving data was read and re-arranged into a form that is more compatible with supervised learning methods. Since the lane change identification problem is a classification problem, a supervised learning method with input-output pairs seems suitable. However, the driving data from UDrive is time-series data. This was problematic because multiple lane changes can occur within a single segment and each of these lane changes may have different patterns of signals. Thus, not only will a single input-output pair for each segment not work, but they do not make any sense either as a segment might be labeled "Lane Change" with no details on the amount of lane change maneuvers performed and when they started. It is necessary to break each segment into several time windows. As for the size of the window, both Mandalia and Salvucci (2005) and Dogan, Edelbrunner, and Iossifidis (2011) achieved the best result when they used between 2.0 and 2.5 seconds. Thus, 2.0 seconds was chosen to be the window size used in this project. The re-arranging process was done with the help from a function proposed by Brownlee (2017) which breaks a time-series data into several constant-sized time windows, making it compatible with supervised learning methods.

Next, a cleanup process was performed to the transformed data. It removed segments which have invalid values that might exist inside time windows due to sensor malfunctions. After cleanup, the next step was to create new class labels for each window based on the E_LaneChange signal. Initially, two classes, "Lane Change" and "No Lane Change" were considered for this problem. However, the data between these two became similar from each other, specifically at the beginning and at the ending of a lane change maneuver. Because of this, it would be harder to make a distinction between the two. Since E_LaneChange signal only had zeros and ones as its value, a ratio between them was calculated and used for categorize the data into four classes:

1. "No Lane Change" (noLC): E_LaneChange signal emits zero for the entirety of the time window. This indicates during a window of two seconds, no lane change maneuver is detected, and driver mostly following the lane in this window.

2. "Lane Change Start" (sLC): E_LaneChange signal begins to emit one and the ratio of ones and zeros is less 0.33. This represents the first 0.67 seconds of the lane change.

3. "Lane Change" (LC): E_LaneChange signal continues to emit 1 after exceeding the ratio of 0.33 as mentioned above, and continues for the duration of lane change. Once it starts to emit 0, the window will still be regarded as LC as long as the ratio of 0's and 1's does not exceed 0.33. This represents the actual process of lane change maneuver.

4. "Lane Change End" (eLC): E_LaneChange continues to emit 0 after exceeding the ratio of 0.33. This represents the ending of a lane change maneuver and the driver is about to complete the lane change and following the lane again.

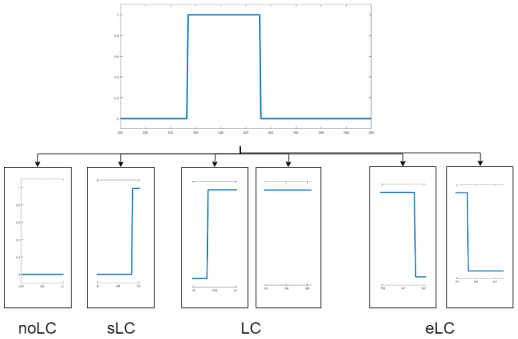

Figure 3.2.1: *How a lane change signal is broken down into four classes*

Approximately 296 segments with a total of 948 lane changes and 16 hours of driving time were processed. Around 493, 060 overlapping time windows were generated. Eighty percent of these used as the training dataset, while twenty percent was set aside to be the test dataset which was not used in the training process.

3.2.4 Statistical Analysis

As it can be seen in the previous section, creating new label relies heavily on the ratio of 0's and 1's. Choosing a wrong ratio can be a contributing factor to an imbalanced class problem. Therefore, a statistical analysis was performed on the training dataset, specifically the student's t-test. T-test was conducted by taking non-overlapping time windows from noLC class and the first window of sLC class for each lane change. Then these two sets were fed into a function TTEST_IND (Jones, Oliphant, Peterson, et al., 2001–) to calculate a p-value for each signal. If one of the p-values was below 0.05, then noLC and sLC class have a significant different between each other.

Imbalanced Class

In total, 394, 448 time windows were selected randomly as the training dataset. Unfortunately, these windows were not balanced in terms of classes. Class noLC was dominating the dataset with 80.68%, followed by class LC having 16.7%, class eLC 1.8%, and lastly class sLC 0.81%. This is expected since lane change maneuvers are not executed very frequently in normal driving conditions. However, this also posed a problem in the training step. If a model was trained using this dataset, the result would be heavily biased towards the noLC class and the model would have a hard time to distinguish the sLC and eLC class.

Two methods were reviewed to alleviate this problem: random undersampling and oversampling. Random undersampling involves taking the majority class and randomly discards data, with each data has the same probability to be discarded, until the number of data is below a certain threshold. On the other hand, oversampling involves replicating data from the minority class until it is above a certain threshold. Excessive oversampling means the minority class will be filled with the same data over and over, and this is detriment to the model training because doing so will induce overfitting in the model. Additionally, data from the noLC class was mostly straight driving data and had similar signal values between each other. Thus, random undersampling on the noLC class was the better choice based on this observation.

3.3 Neural Network Model

3.3.1 Model Construction

Three neural network architectures were used to construct classification models. One model was inspired from Dang et al. (2017) and serves as the base model. The other models were modifications of the first (base) model by adding layers and using different configurations.

Baseline LSTM Model

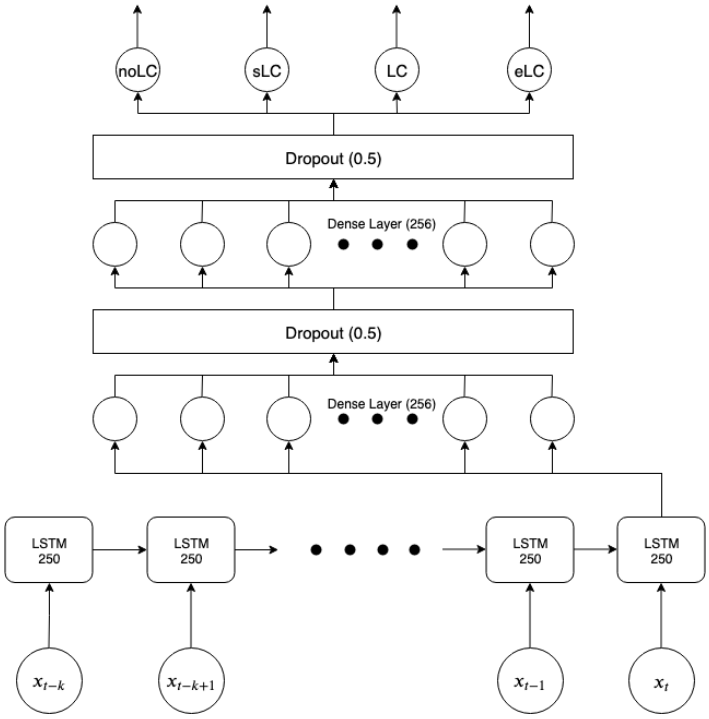

Figure 3.3.1: *A neural network model adopted from Dang, Fürnkranz, Biedermann, and Hoepfl (2017).*

This model was originally proposed to predict the time-to-lane-change (TTLC) by Dang et al. (2017). It utilized five hidden layers. It begins with a single LSTM layer with 250 neurons and output was generated from the final timestep. The output was fed into two fully connected layers with each having 256 neurons. Two dropout layers with 0.5 probability of dropouts were applied between each of the FC layer. A softmax layer was added as the final layer to conform with the defined classification problem. In total, there were $390,076$ trainable parameters in this model.

Some changes were made to adjust this model to be more compatible with the current problem. The model was compiled using RMSprop as the optimizer with learning rate of 0.0005 and categorical cross-entropy as the loss function.

Modified LSTM Model

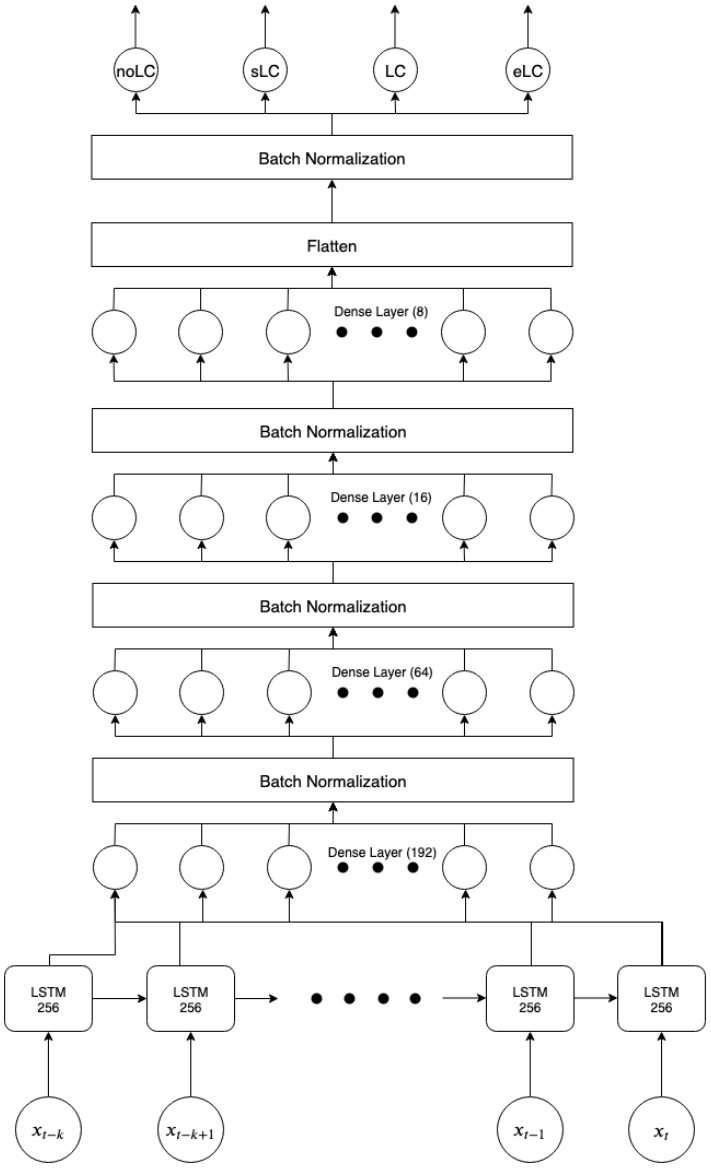

Figure 3.3.2: *A modified neural network based on the previous network. This modes has a deeper fully connected layer compared to the LSTM model.*

A modified LSTM Model was constructed based on the previous model. It employed a single LSTM layer with an increased cell amount, from 250 to 256, and outputs were generated from every timestep. These outputs were put through four layers of a fully connected network with batch normalization applied between each of these layers effectively replacing the dropout layer from the previous model. Since outputs were generated from every timestep, a flatten layer was used before the final batch normalization layer to aggregate outputs across timesteps, thus helping the model to determine the final output. A softmax layer was added as the final layer, similar to the previous model. There were $335,740$ trainable parameters which are slightly less than the baseline model. This was caused by the reduction of the neurons used in the fully connected layer as a compensation for making it deeper.

All configuration parameters remained the same as for the base model. The modified LSTM Model used RMSprop as its optimizer with a learning rate of 0.0005 and categorical cross-entropy as the loss function.

Stacked LSTM Model

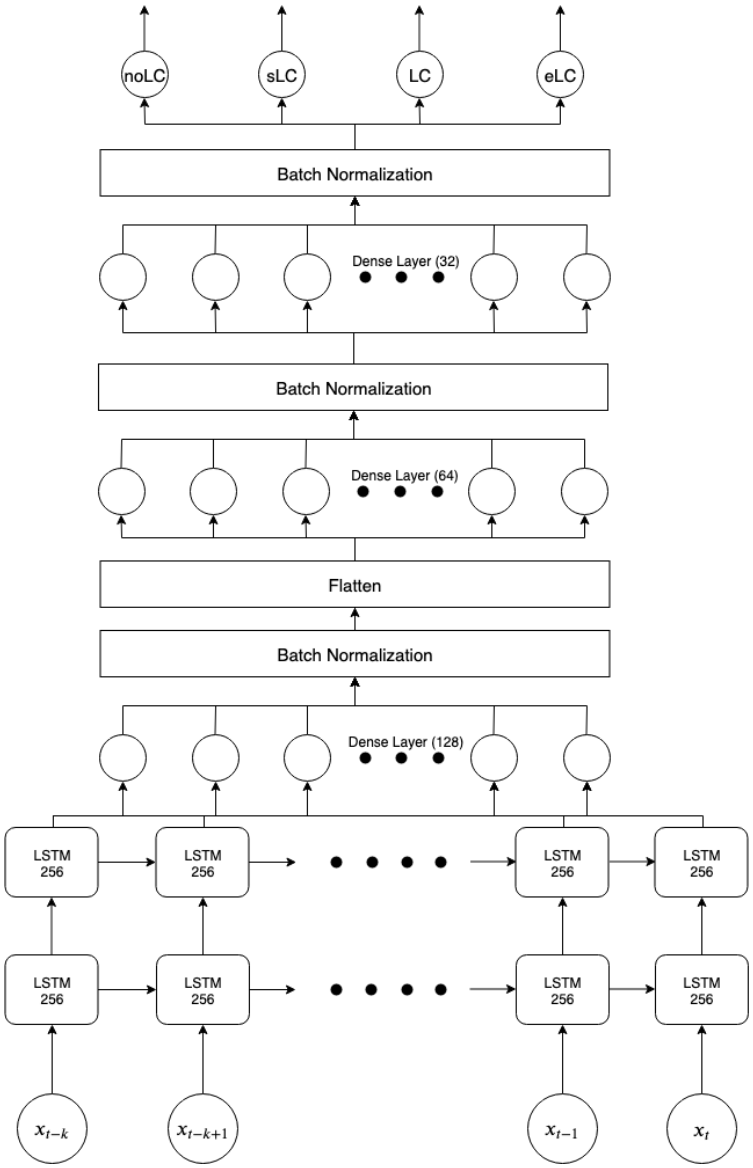

Figure 3.3.3: *Another modification based on the LSTM model. This one has two stacked LSTM layer with a slightly deeper fully connected layer compared to the LSTM layer.*

The stacked LSTM Model was a simpler modification based on the first (base) model. As Figure 3.3.3 shows, this model employs two stacked LSTM with the same 256 cells and three layers of fully connected network. Batch normalization was applied between each layer as a replacement of the dropout layer, similar to Modified LSTM Model. However, a flatten layer was used directly after the first batch normalization, thus the aggregation process occurred earlier than the previous model. Both input and output layers were again identical to the previous model. This model boasted $996,132$ trainable parameters, more than twice as many compared to the previous models. This was caused by the second LSTM layer as the second LSTM layer accommodates every output from every timestep in the first LSTM layer.

Similar to the previous model, this model used RMSprop as the optimizer with the same learning rate of 0.0005. Categorical cross-entropy was used as the loss function.

3.3.2 Hyperparameter Tuning

To find the optimal performance for training and testing, both the modified LSTM and the stacked LSTM model were subjected to different settings of their hyperparameters. These include changing the number of neurons on each layer and applying dropouts where it is applicable. The number of neurons and the dropout probability were increased by a constant interval with each layer having different intervals. After these changes, training was conducted for 50 epochs and the resulting performances were compared with each other.

There were seven sets of hyperparameter changes for the modified LSTM model: three changes in the LSTM layer which involved the number of neurons, the dropout probability, and the recurrent dropout probability, and four changes in each of the fully connected layers which involved the number of neurons. However, for the stacked LSTM model, there were only five sets of hyperparameter changes: the dropout probability and the recurrent dropout probability in the LSTM layer, and three changes in each of the fully connected network. The details can be seen from the table below.

	Hyperparameter Changes	Values
Set #1	LSTM neurons	100, 200, 300
Set #2	First FC neurons	160, 192, 224
Set #3	Second FC neurons	80, 96, 112
Set #4	Third FC neurons	24, 32, 40
Set #5	Fourth FC neurons	12, 16, 20
Set #6	LSTM Dropout	0.3, 0.6, 0.9
Set #7	LSTM Recurrent Dropout	0.3, 0.6, 0.9

Table 3.1: Hyperparameter tuning sets for modified LSTM

	Hyperparameter Changes	Values
Set #1	First FC neurons	128, 192, 256
Set #2	Second FC neurons	64, 96, 128
Set #3	Third FC neurons	32, 48, 64
Set #4	LSTM Dropout	0.3, 0.6, 0.9
Set #5	LSTM Recurrent Dropout	0.3, 0.6, 0.9

Table 3.2: Hyperparameter Tuning for Stacked LSTM

3.3.3 Model Training

Training process was conducted using an identical configuration: 500 epochs, i.e. loops, with batch size of 64. Three Keras specific callbacks were used and these were ModelCheckpoint, EarlyStopping, and CSVLogger. ModelCheckpoint was used to guarantee the best performing model during training process would be saved, EarlyStopping was used during the hyperparameter tuning to stop the training process if the loss value was not improved until a certain epoch had passed, and CSVLogger was used to log the training performance into CSV files.

During training, 20% from the training data were held back and served as a validation set while the remaining 80% were used as the actual training data. After each epoch, the model was checked using the validation set. Parameters within the model were tuned based on the performance of the validation process. Once training was finished, four metrics were extracted: training accuracy, training loss, validation accuracy, and validation loss. These four values were used to determine whether a model achieved convergence and fit for the testing phase, or if it suffered from overfitting or underfitting.

4 Results

As stated in the previous chapter, training was conducted for the proposed networks with the same configuration and the same dataset. Afterwards, they were subjected to testing with the same test dataset. This way, the result can be compared across neural network models. Both the neural network performance figures and the hyperparameter tuning results are given in this chapter. The results are given in graphs which depict the

training performances. Additionally for the neural network performance, confusion matrices and F1 scores are also provided. The numbers that shown on the confusion matrices represents the time windows as mentioned in Chapter 3 Section 3.2.3.

4.0.1 Neural Network Performances

The results for the neural network performances are grouped by model. Each model has training accuracy and loss graphs, and a confusion matrix which denotes their testing performance. F1 scores of these three models were compared to each other with both micro and macro averages. The cross validation evaluation was conducted only for the stacked LSTM model. Since the F1 score was used in the cross validation, its value is divided into micro and macro averages as well.

Baseline LSTM Model

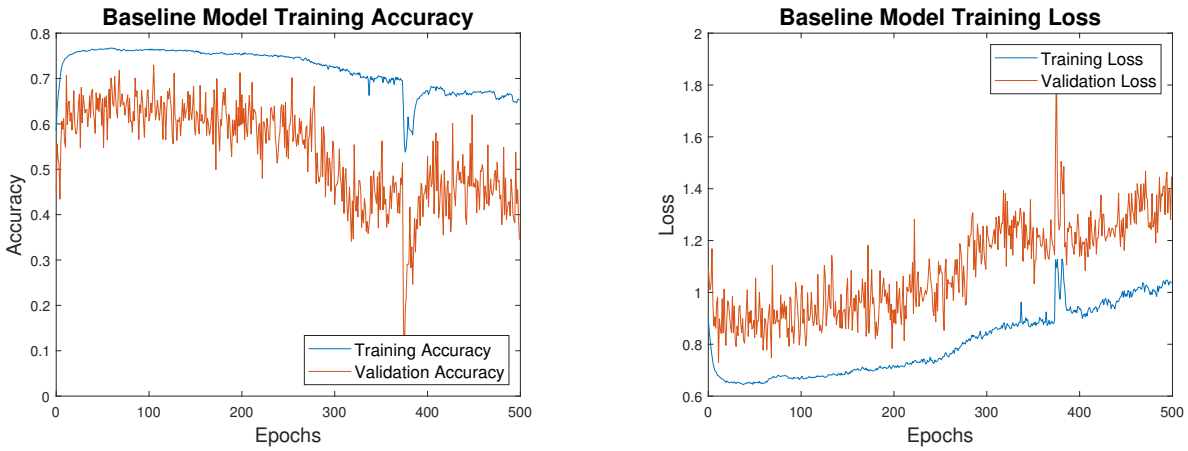

Figure 4.0.1: *Training performances of the baseline model.*

Training performances for the baseline model can be seen from Figure 4.0.1. The left and right figures are the training and validation accuracy and loss for each epoch, respectively. It can be seen from the figures that the baseline model was struggling during training phase. The training accuracy hits around 75% mark and it started to go down as the training went on. On the other hand, validation accuracy was fluctuating widely between 30% and 70%, although in some cases it plummeted down to 10%. Both the training and validation accuracy had a downward trend which means it got worse over time. The same can be said when looking at the loss graph. The training loss went down at the beginning, but after 100 epochs it began to increase steadily. This means the training performance becoming worse as the training went on. The validation loss has similar characteristics with the validation accuracy: fluctuating wildly and it got worse over time. Both the training and validation loss had an upward trend.

After the training phase was done, the baseline model was evaluated with the test dataset. The evaluation result can be seen in Figure 4.0.2. It shows that the baseline model labeled every test data as either "noLC" and "LC" only. By doing this, the model managed to identify the majority of "noLC" and "LC" class. However, this also means that the model completely missed the "sLC" and "eLC" class, indicating there is something wrong with the training process. As a result, the model has a disparity between F1 micro and F1 macro, which will be presented later. In summary, this model was not very good at detecting lane changes.

Modified LSTM Model

As it can be seen from Figure 4.0.3, the modified LSTM model was slightly better than the baseline model in terms of training performance. It can be seen that the training accuracy and training loss were getting better as training went on. This is normal as during the training phase, both training accuracy and training loss were expected to be better, approaching 100% accuracy and zero loss, respectively. However, the model has several problems. Firstly, it was slow. After 500 epochs passed, it only managed to achieved approximately 80% in

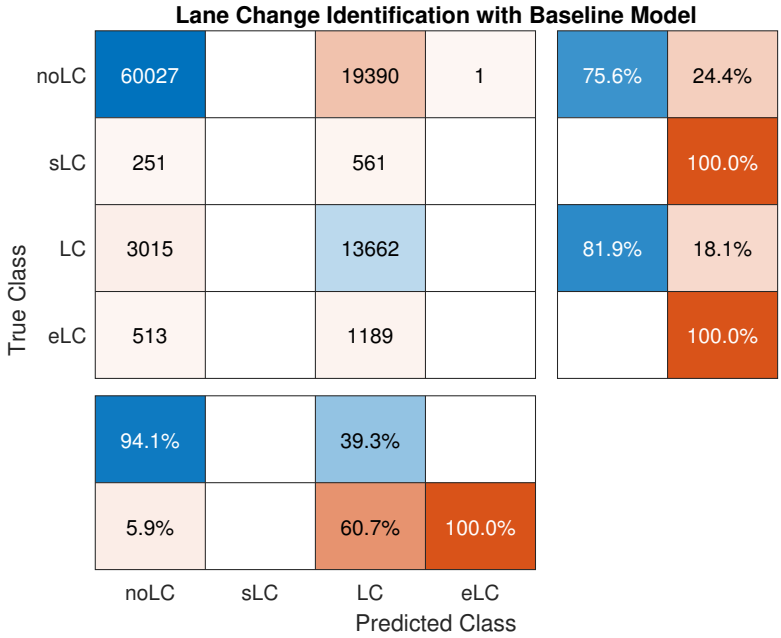

Figure 4.0.2: *The confusion matrix for the Baseline LSTM model*

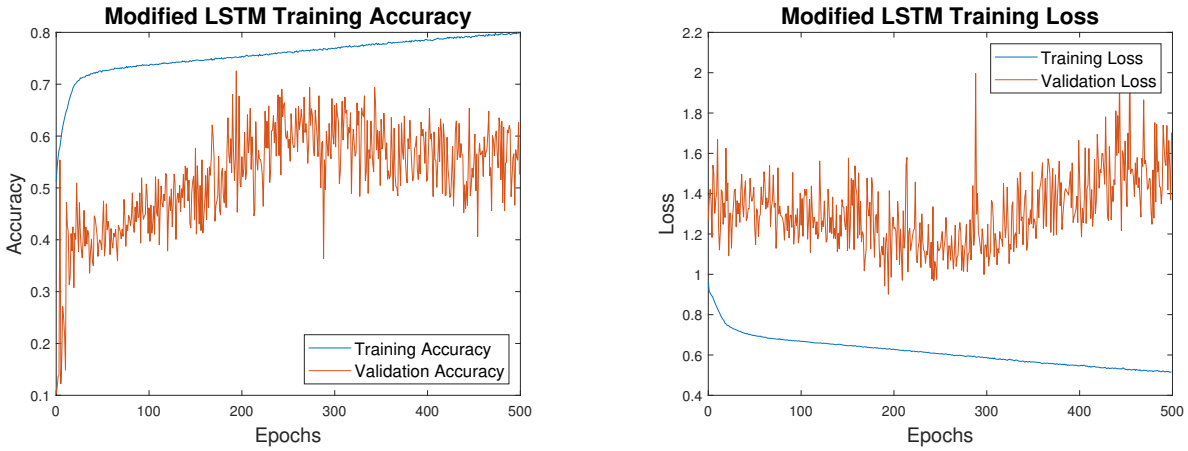

Figure 4.0.3: *Training performances of the modified LSTM model.*

training accuracy and 0.6 in training loss. Secondly, the validation accuracy and loss were also fluctuating widely. Thirdly, its validation loss were worse than the baseline model's training loss, reaching a maximum of 2.2 of loss. Best validation accuracy and loss were achieved before 300 epochs passed and after that point, it steadily went worse.

When the modified LSTM model was put to test using the same test dataset, the resulting confusion matrix can be seen from Figure 4.0.4. It shows that while the modified LSTM model tried to identify every class, its performance was not good. While it managed to identify the majority of the "LC" class, i.e., 84.1% of them, it also made a bad guess on every other class. Specifically, 68.4% of the "noLC" class were classified as "LC", which is highlighted in red in the figure, only 4% of the "eLC" class were identified, and less than 1% of the "sLC" class. These misclassifications have a large negative effect on its F1 scores, both in micro and macro average as shown later.

Stacked LSTM Model

According to Figure 4.0.5, the training phase was running well for the stacked LSTM model. It managed to achieve 95% of training accuracy and almost 0.1 of training loss. Moreover, its validation accuracy and loss

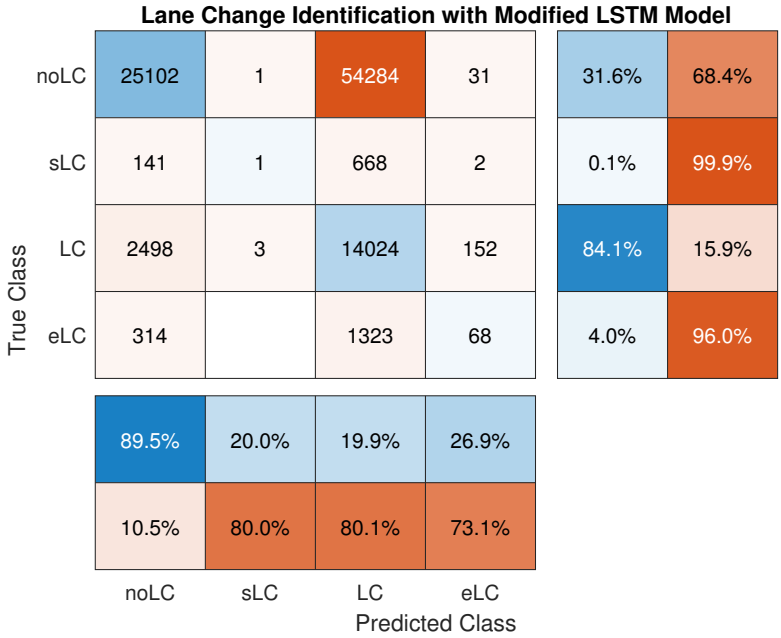

Figure 4.0.4: *The confusion matrix for the Modified LSTM model*

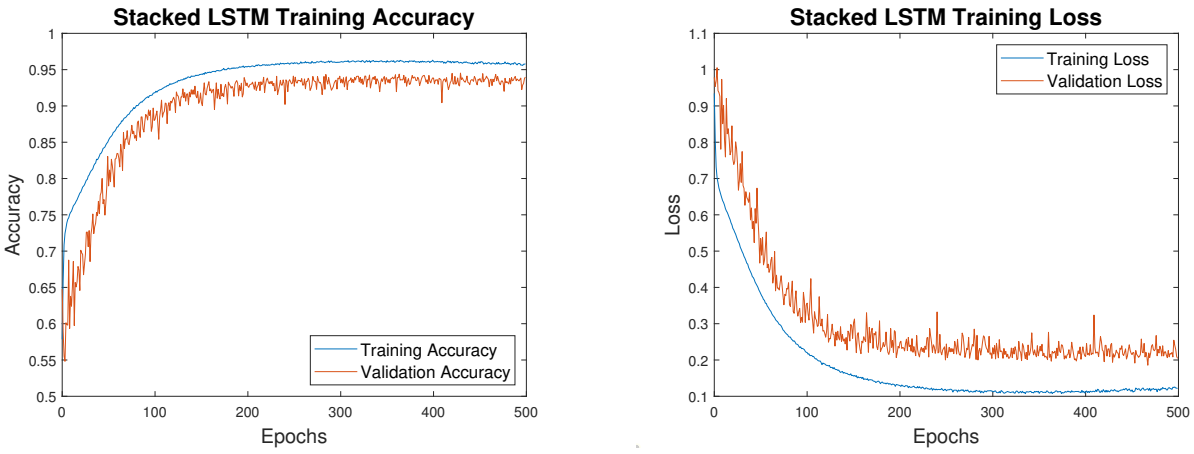

Figure 4.0.5: *The training performances of the stacked LSTM model.*

were still fluctuating, but they were significantly better than the other models because they showed an upward trend and downward trend for the accuracy and loss respectively. At the end of the training, they achieved approximately 90% in validation accuracy and 0.15 in validation loss. However, this model has one problem. When the loss graph is examined closely, one can see that the training loss were increasing slowly after 300 epochs. This indicates the model started to suffer from overfitting, although the effects were still minor.

Testing was conducted on the stacked LSTM to judge its final performance, and it produced a confusion matrix which can be seen in Figure 4.0.6. This is generally better than the other models as it identified the majority of "noLC" and "LC" classes correctly while also identified a decent amount of "sLC" and "eLC" classes. Specifically, 90.9% of the "noLC", 59.2% of the sLC, 95.8% of the "LC", and 77.2% of the "eLC" were correctly identified. It can also be seen that the incorrect guesses were sparsely distributed with most of them were made on "sLC" and "eLC" classes. These misclassifications have a negative effect on its F1 score, although it is not as severe as the modified LSTM model.

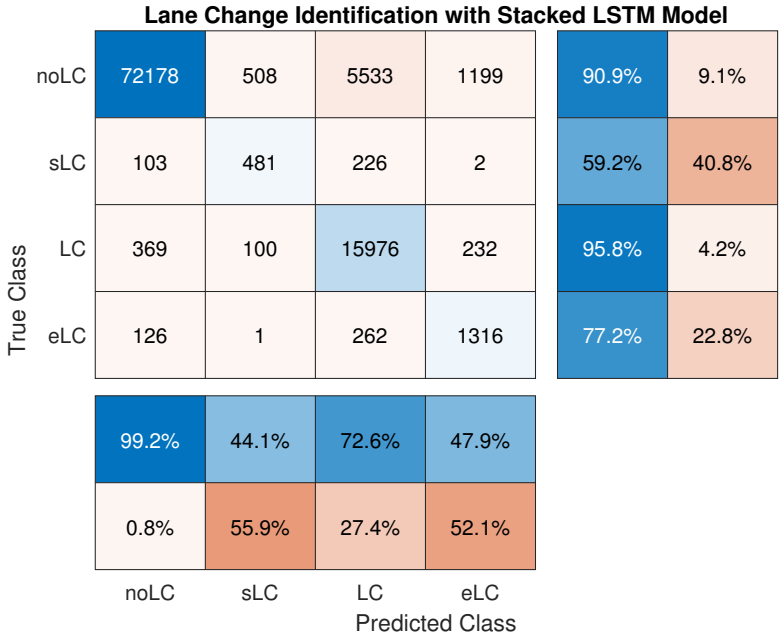

Figure 4.0.6: *The confusion matrix for the Stacked LSTM model*

F1 scores

To have a better grasp on the performances of these models, F1 scores were calculated using both micro and macro averages. Both confusion matrices and F1 scores were used to evaluate the performance of the models.

	Baseline LSTM	Modified LSTM	Stacked LSTM
F1 Score Micro	0.7473	0.3975	0.9122
F1 Score Macro	0.3431	0.2154	0.7178

Table 4.1: F1 Scores between each model

As Table 4.1 has shown, the best F1 score in both micro and macro average was achieved by the stacked LSTM model which corresponds to its confusion matrix. Conversely, the modified LSTM had the lowest F1 score in both micro and macro average. However, it was not enough to measure the general performance of the stacked LSTM model because the test data was selected randomly and it might be similar to the training data. Thus, cross validation evaluation was performed on the stacked LSTM model. During this evaluation, F1 scores were calculated for each fold, and the final score was produced by calculating the average of F1 scores across the folds. As it can be seen from Table 4.2, the F1 scores in both micro and macro average have a rather small margin between the folds. This indicates the general performance of the stacked LSTM model is somewhere around these values.

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Average
Stacked LSTM (Micro)	0.9157	0.9209	0.9119	0.9202	0.9215	0.9180
Stacked LSTM (Macro)	0.6879	0.6876	0.6692	0.6964	0.7023	0.6887

Table 4.2: F1 Scores for each fold in cross validation process

4.0.2 Hyperparameter Tuning

Both the modified LSTM and the stacked LSTM were subjected to the hyperparameter tuning process in order to find the optimal setting that can draw the best performance for them. As it has been explain in the previous chapter, hyperparameter changes were divided into several sets and each of them have their own impact. Each set was represented by a training loss graph as it is enough to approximate the general performance of a model.

Modified LSTM model

As it can be seen from Figure 4.0.7, several hyperparameter changes were tested. The set #1, which changed the number of neurons in the LSTM layer, was the only one which showed an improvement, as the loss decreased when the LSTM had 200 and 300 neurons compared to just 100 neurons. However the difference between using 200 and 300 neurons were small, indicating that the ideal neuron was somewhere between these two numbers. The set #2 to #5, in which the number of neurons was changed in each fully connected layer, produced roughly the same result. Generally, the more neurons they had, the better they were, but the improvements were small. This can be seen easily in set #5 as changes in the fourth fully connected layer didn't have any significant changes to the training loss. The set #6 and #7, which changed the dropout probability in the LSTM layer, were the only one which produced varied results. In set #6, the model achieved overall the best loss when using a dropout probability of 0.6. However, training stopped after 20 epochs because of the EarlyStopping callback. Dropout probability of 0.3 produced a slightly worse result, and dropout probability of 0.9 produced the worst result. In the final set, their loss was increased as the recurring dropout probability was also increased.

Stacked LSTM model

Five sets of hyperparameter changes were tested, three of them changed the number of neurons in the fully connected layers while two of them changed the dropout and recurrent dropout probability of the LSTM layers. The results can be seen in Figure 4.0.8. In general, immediate improvements are clearly visible with all the sets. However, all of them also triggered the EarlyStopping callback because training went on without improving the best loss. Thus, none of them finished the training process. Furthermore, while they showed a downward trend at first, they stagnated after 10 epochs. When compared to the original setting, at first the original setting had the worst performance. However as the training went on, it surpassed the others and still continued to be better.

5 Discussion & Conclusion

Three neural network models were constructed with different architectures and hyperparameters. These models were trained with the same dataset and configurations, and they evaluated with the same methods.

The results show that using neural network approach to identify lane change maneuvers in UDRIVE is possible, particularly with the stacked LSTM model. This model performed better than the baseline and the modified LSTM model. However, the stacked LSTM model still has room for improvement as misclassifications were still made even though the training went well. As for the other models, they struggled during the training and test process. This is due to several factors, which will be discussed below.

5.1 Neural Network Models

5.1.1 Baseline Model

As it has been stated in Chapter 1 and 3, the baseline model was adopted from Dang et al. (2017). This model was chosen because it used LSTM, which is highly compatible with time-series data, and it reaches a desired layer depth. It has one LSTM layer and two fully connected layers, making it categorized as a deep neural network. The same parameters were applied here with exception of the optimizer and the learning rate. Variables used for training are already mentioned in the Chapter 3 Section 3.2.2. However compared to what was originally used by Dang et al. (2017), gazing behaviour and head rotation was not used in both the training and testing process.

The results show that the baseline model struggled during training and testing phase. Normally, a neural network is expected to have high enough accuracy, i.e., above a certain value, e.g., 95%, and low loss at the end of the training phase. However, as the training went on for this model, it got worse over time in both the accuracy and the loss. During testing, it also labeled the data only as either "noLC" or "LC" while ignoring the "sLC" and "eLC" (start and end of lange change). This was caused by the fact that the training dataset was heavily dominated by "noLC" and "LC" classes, suggesting that the undersampling might not be aggresive enough. The "sLC" and "eLC" were minuscule compared to the other two. This, along with the training

struggle, caused the model to ignore "sLC" and "eLC" as it could not distinguish any pattern which is unique to these classes.

Although only "noLC" and "LC" were found for this model, these two classes were identified with rather good performance which is why their F1 score micro is quite high. F1 scores with micro average takes class imbalance into the account and as stated above, both "noLC" and "LC" data is dominating. Thus, one can easily achieve high accuracy just by predicting these two classes and ignoring the rest. On the other hand, F1 macro score assumes every class has the same weight, hence ignoring class imbalance. Since the model did not make any guess in both "sLC" and "eLC", the F1 macro score suffered heavily.

According to Griesbach (2019), driver gaze behaviour and head rotation could be a good indicator for lane change identification. Not having these variables was proven to be detrimental in lane change identification as these two variables were not part of the input data. Moreover, the LSTM architecture might also contribute to the difficulties in the training. By having only the last timestep to transmit output to the next layer in the LSTM, some information might be lost in the process. Another plausible factor is the number of parameter in this model. It has approximately 390.000 trainable parameters and while this might seems to be sufficient, the results show that this number should be changed. Increasing the number of trainable parameters might lead to a better performance with the risk of overfitting, while decreasing them might lead to a faster training time with the risk of underfitting.

5.1.2 Modified LSTM Model

In the modified LSTM model, two changes were made compared to the base model. Outputs were emitted for each timestep in the LSTM layer instead of only at the last. This way, information is preserved between timesteps, and thus it can be taken into consideration for the next layer. Furthermore, the depth of the fully connected layer was also increased as a deeper fully connected layer often means a performance improvement. However, the number of neurons was reduced to compensate the depth as going too deep with too many neurons often results in overfitting. As a result, the number of trainable parameters in this model is less than the baseline model.

The results show that this model is better than the baseline model during the training phase. The increase in the depth of the fully connected network may have helped during the training phase as both training accuracy and training loss got better over time. However, both the validation accuracy and loss were worse than the baseline model. Moreover, when this model was put to test, it achieved worse F1 scores than the baseline model. It is believed that the low scores was caused by the low number of parameters in this network. Having too few parameters often means the model could not capture the patterns of a lane change maneuver. Moreover, both the training accuracy and loss converged less than what was expected, signifying that this model might be suffering from underfitting.

To alleviate this, several hyperparameters were changed and tested to make the model perform better. The changes were mostly in the number of neurons in each layer to increase its trainable parameters. Dropout was also applied in the LSTM layer. Some of these changes resulted in immediate improvement at the start of the training compared to the original setting. However, these changes also produced similar results. They all seems to converge around 0.45 loss, and even in some hyperparameter changes, like the dropout and the recurrent dropout, the loss is even higher than before. Thus, it is believed that making the fully connected layer deeper was not enough to improve the performance. By looking at the model structure, it is clear that regardless of the depth of the fully connected layer, they still depend on a single LSTM layer because it interacts directly with the input data. If the LSTM layer made a mistake, the fully connected layer will base their choice on this mistake, and this can be detrimental to the overall performance.

5.1.3 Stacked LSTM Model

Based on the problems of the modified LSTM model, this model has two LSTM layers stacked on top of each other. The second LSTM layer exist to review every output and minimize any mistake made by the first LSTM layer. Furthermore, this model has an extra fully connected layer, making it slightly deeper compared to the baseline model but not as deep as the modified LSTM layer. These modification results in a massive increase in the number of trainable parameters, almost three times the number of trainable parameters in the baseline model.

The results show that this model performed much better compared the other two models. This is due to the high number of trainable parameters in the model caused by the second LSTM layer. Trainable parameters

determine the ability of a neural network to learn patterns within the input data. More trainable parameters often mean better ability to distinguish complex patterns, but too much of them can make a network suffers from overfitting. In this case, the stacked LSTM model certainly benefited with its high number of parameters, but it also suffers from minor overfitting. This means either it has too many trainable parameters, or it is trained for too many epoch.

Like the modified LSTM model, hyperparameter tuning was also conducted for this model. The number of neurons of each fully connected layer was changed and dropout also applied on the LSTM layers. It produced an immediate improvement, but all of them could not finished the training process as they stopped earlier than expected. Moreover, their loss converged to a loss that is worse than the original setting. It is suspected that these changes result in overfitting as in their original setting, the number of parameters were already high, almost three times as the number of parameters in the baseline model.

Aside from overfitting, the stacked LSTM model also has some issues. Firstly, the model has a mediocre performance according to its F1 scores macro, although it was the best when compared with the other models. Increasing the number of parameters would not work because it already shows signs of overfitting, so model improvement would be tricky. Secondly, the model takes a long time to train due to the large number of trainable parameters, so it can be difficult to tune the hyperparameters since training would have to start from scratch.

5.2 General Performance and Limitations

As it has been shown, these networks have a highly varied performances, with the baseline model performed the worst and the stacked LSTM performed the best. However, the stacked LSTM model produced a rather mediocre results. Several factors which affect the performance were identified and they have been consistent across the models. These factors are the input variables selection, the architecture, the number of trainable parameters, and their hyperparameters. Other factors also contributes to the performance, such as how the input data is processed.

The training dataset used by the models only consists of vehicle data and their surrounding environment. Head rotation and gaze direction only exists in the form of video, so, as of this writing, they cannot used directly by the neural network. Griesbach (2019) mentioned that these two variables could be a good indicator to identify a lane change and Dang et al. (2017) used them in their network and produced good results. Without these two, identify lane change was more harder, as evidenced by the results. Furthermore, the training dataset suffers from imbalanced data and, unfortunately, this was expected as it has been explained in Chapter 3 Section 3.2.4.

The neural network architecture and the number of trainable parameters are also directly affecting its performance, as evidenced by the results. Since the data is in the form of time-series data, using LSTM as the first layer is logical because it can use the information from the previous timestep to influence the current output. The LSTM layer were connected to several fully connected layers. These layers acts as the classifier, i.e., they decide which class the data belongs to. All of these models proposed here were using this combination and their results were highly varied. They both have around the same amount of trainable parameters, but it seems this was not enough because they were struggling during both training and testing phase. The modified LSTM even have a deeper fully connected layer than the baseline model, thus hinting at the modification should be done in the LSTM layer instead. When another LSTM layer was added, the model suddenly have a much higher trainable parameter, almost three times as much compared to the other two. As a result, it excels in training compared to the other two. However, the testing results only got slightly better. This indicates the stacked LSTM model shows promise, but it still needs an improvement. At this stage, the improvement is usually done in the form of finding the optimal hyperparameters, such as the number of neurons of each layer, which activation functions should be used, which optimizer should be used, how much is the learning rate, etc.

It might also be the case that the hyperparameter tuning process was not enough. In this project, hyperparameter tuning only affects the number of neurons in each layer and the LSTM's dropout probabilities. There are other hyperparameter that can affect a neural network's performance, e.g., the choice of the optimizer function, the learning rate, the number of epochs, the number of batches, etc. Considering the possibilities, the number of hyperparameter changes in this project are certainly limited as it only changed by a fixed amount. One solution of testing more hyperparameter changes would be to use a grid search algorithm or even employs an evolutionary algorithm, although the training time will then increase exponentially.

Another factor which can influence neural network performance is how the input data is processed. In

this case, raw input data was broken down into several time windows of two seconds and labels were given by looking at the ratio of zeros and ones inside a time window. These two processes affects the performance of the model. Firstly, the length of the time window controls how much data will be fed into the model. As a result, the model has more information and it can make different judgments, thus leading to a different performance. Secondly, the ratio controls the class distribution of the data. Given the right ratio, data inside a class should be significantly different from the other class. Consequently, the model should be able to identify each class more clearly and make a better guess. Finding the optimal setting for these two can be difficult to do. When changing the time window length, the models will have to be adjusted to accommodate the change, and training will have to start from scratch. On the other hand, changing the ratio can produce an imbalanced training data. Assuming the time window is still the same, the model need no adjustment. However when trained on this data, the model may not perform as expected i.e., it might be biased toward one or more classes just as reflected in the training data.

As it was mentioned in Chapter 3 Section 3.3, the baseline model was adopted from Dang et al. (2017) because they reported a good performance while using this model to classify lane change maneuver. However when it was applied using UDRIVE as the training data, it performed well below what was reported. It is already mentioned that this might be caused by omitting head rotation and gaze direction variables. Thus, the stacked LSTM model has a potential to be better than the model from Dang et al. (2017) if given these variables. Moreover if the other issues are fixed, humans can use this model to identify lane change maneuvers quicker and more accurately compared to the manual methods. This model can also potentially be used by ADAS directly to improve their modules which relate to lane change maneuvers, such as lane keeping system, lane departure warning system, lane change assistance system, etc. As a result, drivers may perform lane change maneuvers more safely and thus, accidents can be averted.

5.3 Conclusion

Identifying lane change maneuver in highway driving can be done using a deep neural network. Three neural network models were implemented to identify lane change maneuvers using naturalistic driving data from the UDRIVE project. All three models were using a combination of LSTM and fully connected layers with each having a different modifications. The baseline model, which adapted from Dang et al. (2017), was performing below expectation since head rotation and gaze direction was omitted. The modified LSTM model, which has deeper fully connected layer than the baseline model, performed slightly better during the training phase, but worse during the testing phase. The stacked LSTM model, which stacked two LSTM layers and had a slightly deeper fully connected layer than the baseline model, performed the best during training phase. It also performed the best during testing phase, however it was still not good enough as it had only an F1 score of 0.6887 on average when evaluated using cross-validation. Several factors were identified as the reason why the model rather struggling with identifying lane changes, especially with the start of lane change. These factors are: the time window size and the labeling ratio, the variable selection and the imbalanced data, and the neural network architecture, their hyperparameters, and the number of trainable parameters. Despite this, the stacked LSTM model seems promising and once its issue is fixed, it should be able to identify lane change maneuver easily.

6 Future Work

Future work of this project should focused on improving the performance of the stacked LSTM model. This can be done in many different ways. One idea would be to merge the outputs from the first LSTM layer into the next layer. By merging outputs across timesteps, information are consolidated and as a result, the second LSTM layer will be fed with less timesteps, thereby reducing the number of parameters. Another idea would be to implement another neural network to find the head rotation and gaze direction from the video input. The model's performance can be better if these two variables is available. Choosing an optimal configuration for a neural network model can also be better by using grid search method or employing evolutionary algorithms, such as genetic algorithm or particle swarm optimization. Using them might mean the training time will be increased considerably, but when they have finished, the optimal configuration is found. Once a good model with an optimal configuration is found, it can be applied to the UDRIVE data to help with the lane change identification. A comparison test can also be conducted between lane change identification with neural network

and lane change identification with manual video recognition. Thus, the performance gap can be discovered and used further to improve the development of the neural network.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {usenix} symposium on operating systems design and implementation ({osdi} 16)* (pp. 265–283).

Abrahamsen, E., Brastein, O. M., & Lie, B. (2018). Machine learning in python for weather forecast based on freely available weather data. In *Proceedings of the 59th conference on simulation and modelling (sims 59), 26-28 september 2018, oslo metropolitan university, norway* (153, pp. 169–176). Linköping University Electronic Press.

Adley Law Firm. (2019). 25 reasons car accidents happen & how to prevent them. Retrieved from https://adleylawfirm.com/25-reasons-car-accidents-happen-and-how-to-prevent-them/

Ahlswede, A. (2018). Entwicklung der gesamten fahrleistung von kraftfahrzeugen auf autobahnen in deutschland von 1990 bis 2017 (in milliarden kilometer). Retrieved from https://de.statista.com/statistik/daten/studie/155732/umfrage/fahrleistung-auf-autobahnen-in-deutschland/

Ayalasomayajula, V. (2016). Visualizing time series data: 7 types of temporal visualizations. Retrieved from https://blog.socialcops.com/academy/resources/visualizing-time-series-data/

Bärgman, J., van Nes, N., Christoph, M., Jansen, R., Heijne, V., Carsten, O., & Fox, C. (2017). *The udrive dataset and key analysis results.* Tech. rep. 2017.: 10. 26323/UDRIVE.

Barnard, Y., Utesch, F., van Nes, N., Eenink, R., & Baumann, M. (2016). The study design of udrive: The naturalistic driving study across europe for cars, trucks and scooters. *European Transport Research Review* **8**.(2) (2016), 14.

Beggiato, M. & Krems, J. (2013). Sequence analysis of glance patterns to predict lane changes on urban arterial roads. *6. Tagung Fahrerassistenz* **49**.(0) (2013). Retrieved from http://mediatum.ub.tum.de/doc/1187197/1187197.pdf

Bengio, Y., Boulanger-Lewandowski, N., & Pascanu, R. (2013). Advances in optimizing recurrent networks. In *2013 ieee international conference on acoustics, speech and signal processing* (pp. 8624–8628). IEEE.

Bengio, Y., Simard, P., Frasconi, P., et al. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks* **5**.(2) (1994), 157–166.

Brockwell, P. J., Davis, R. A., & Calder, M. V. (2002). *Introduction to time series and forecasting.* Springer.

Brownlee, J. (2017). How to convert a time series to a supervised learning problem in python. Retrieved from https://machinelearningmastery.com/convert-time-series-supervised-learning-problem-python/

Cerdeira, M., Falcón, P., Delgado, E., & Barreiro, A. (2018). Reset controller design based on error minimization for a lane change maneuver. *Sensors* **18**.(7) (2018), 2204.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

Chollet, F. et al. (2015). Keras. https://keras.io.

Chovan, J., Tijerina, L., Alexander, G., & Hendricks, D. (1994). *Examination of lane change crashes and potential ivhs countermeasures. final report.*

Cui, Z., Ke, R., & Wang, Y. (2018). Deep Bidirectional and Unidirectional LSTM Recurrent Neural Network for Network-wide Traffic Speed Prediction (2018), 1–12. arXiv: 1801.02143. Retrieved from http://arxiv.org/abs/1801.02143

Dang, H. Q., Fürnkranz, J., Biedermann, A., & Hoepfl, M. (2017). Time-to-lane-change prediction with deep learning. In *2017 ieee 20th international conference on intelligent transportation systems (itsc)* (pp. 1–7). IEEE.

Dauphin, Y. N., Fan, A., Auli, M., & Grangier, D. (2017). Language modeling with gated convolutional networks. In *Proceedings of the 34th international conference on machine learning-volume 70* (pp. 933–941). JMLR. org.

Deka, L. & Quddus, M. (2013). Network-level accident-mapping: Distance based pattern matching using artificial neural network. *Accident; analysis and prevention* **65C** (2013, December), 105–113. doi:10.1016/j.aap.2013.12.001

Department for Transport. (2018). Road Traffic Estimates : Great Britain 2017. *Department for Transport* **2014**.(July) (2018), 37. Retrieved from https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment%7B%5C_%7Ddata/file/741953/road-traffic-estimates-in-great-britain-2017.pdf

Ding, C., Wang, W., Wang, X., & Baumann, M. (2013). A neural network model for driver's lane-changing trajectory prediction in urban traffic flow. *Mathematical Problems in Engineering* **2013** (2013).

Dogan, Ü., Edelbrunner, J., & Iossifidis, I. (2011). Autonomous driving: A comparison of machine learning techniques by means of the prediction of lane change behavior. In *2011 ieee international conference on robotics and biomimetics* (pp. 1837–1843). IEEE.

Drakos, G. (2018). Cross-validation. Retrieved from https://towardsdatascience.com/cross-validation-70289113a072

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* **12**.(Jul) (2011), 2121–2159.

European Road Safety Observatory. (2018). Motorways 2018 (2018). Retrieved from https://ec.europa.eu/transport/road_safety/sites/roadsafety/files/pdf/ersosynthesis2018-motorways.pdf

FHWA. (1998). Our nation's highways: Selected facts and figures. *US Department of Transportation, Federal Highway Administration, Publication FHWA-PL-98-015,* (1998).

FHWA. (2016). Annual vehicle distance traveled in miles and related data - 2016 by highway category and vehicle type. Retrieved from https://www.fhwa.dot.gov/policyinformation/statistics/2016/vm1.cfm

Fitch, G. M., Lee, S. E., Klauer, S., Hankey, J., Sudweeks, J., & Dingus, T. (2009). Analysis of Lane-Change Crashes and Near-Crashes. *Final Report DOT HS 811 147, US Department of Transportation, National Highway Traffic Safety Administration* (June) (2009), 1–88. doi:DOTHS811147

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. http://www.deeplearningbook.org. MIT Press.

Griesbach, K. (2019). *Lane change prediction in the urban areas* (Doctoral dissertation, Chemnitz University of Technology).

Guresen, E., Kayakutlu, G., & Daim, T. U. (2011). Using artificial neural network models in stock market index prediction. *Expert Systems with Applications* **38**.(8) (2011), 10389–10397.

Hebb, D. O. (1949). The Organisation of Behavior. *John Wiley & Sons, Inc* (1949), 1–365. doi:10.1364/OL.24.000954

Hill, J. W. (2017). Deep Learning for Emotion Recognition in Cartoons by Project Supervisor : Stefanos Kollias. (June) (2017).

Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural computation* **9**.(8) (1997), 1735–1780.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in science & engineering* **9**.(3) (2007), 90–95.

Ioffe, S. & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

ISO, I. (2005). 9000-quality management systems–fundamentals and vocabulary. *Quality management systems, Berlin: Beuth Verlag GmbH* (2005), 22–25.

Jain, A., Koppula, H. S., Soh, S., Raghavan, B., Singh, A., & Saxena, A. (2016). Brain4Cars: Car That Knows Before You Do via Sensory-Fusion Deep Learning Architecture (2016). doi:10.2766/248698. arXiv:1601.00740

Johnson, J. (2013). General regression and over fitting. Retrieved from https://shapeofdata.wordpress.com/2013/03/26/general-regression-and-over-fitting/

Jones, E., Oliphant, T., Peterson, P., et al. (2001–). SciPy: Open source scientific tools for Python. [Online; accessed ¡today¿]. Retrieved from http://www.scipy.org/

Joshi, R. (2016). Accuracy, precision, recall & f1 score: Interpretation of performance measures. Retrieved from https://blog.exsilio.com/all/accuracy-precision-recall-f1-score-interpretation-of-performance-measures/

Karlik, B. & Olgac, A. V. (2011). Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems* **1**.(4) (2011), 111–122.

Kim, I.-H., Bong, J.-H., Park, J., & Park, S. (2017). Prediction of driver's intention of lane change by augmenting sensor information using machine learning techniques. *Sensors* **17**.(6) (2017), 1350.

Kirkland, G. (2019). Motorways in europe and the uk: Facts, routes, toll roads, and speed limits. Retrieved from https://www.oponeo.co.uk/tyre-article/motorways-in-europe-and-the-uk

Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B. E., Bussonnier, M., Frederic, J., ... Corlay, S., et al. (2016). Jupyter notebooks-a publishing format for reproducible computational workflows. In *Elpub* (pp. 87–90).

Kratzert, F. (2016). Understanding the backward pass through batch normalization layer. Retrieved from https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

Kulkarni, A. (2017). What the heck is time-series data (and why do i need a time-series database)? Retrieved from https://medium.com/timescale/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature* **521**.(7553) (2015), 436.

Leonhardt, V. & Wanielik, G. (2018). Neural network for lane change prediction assessing driving situation, driver behavior and vehicle movement. *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC* **2018-March** (2018), 1–6. doi:10.1109/ITSC.2017.8317832

Lever & Ecker. (2018). Causes of car accidents. Retrieved from https://www.leverecker.com/car-accident-lawyer/causes/

Li, G., Li, S. E., Jia, L., Wang, W., Cheng, B., & Chen, F. (2015). Driving maneuvers analysis using naturalistic highway driving data. In *2015 ieee 18th international conference on intelligent transportation systems* (pp. 1761–1766). IEEE.

Lv, Y., Duan, Y., Kang, W., Li, Z., & Wang, F.-Y. (2014). Traffic flow prediction with big data: A deep learning approach. *IEEE Transactions on Intelligent Transportation Systems* **16**.(2) (2014), 865–873.

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml* (Vol. 30, *1*, p. 3).

Mandalia, H. M. & Salvucci, M. D. D. (2005). Using Support Vector Machines for Lane-Change Detection. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* **49**.(22) (2005), 1965–1969. doi:10.1177/154193120504902217

Martens, J. & Sutskever, I. (2011). Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th international conference on machine learning (icml-11)* (pp. 1033–1040). Citeseer.

McCulloch, W. & Pitts, W. (1943). A logical calculus of the ideas imminent in nervous activity. *Bulletin of Mathematical Biophysics* **5** (1943), 115–133.

Nair, V. & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (icml-10)* (pp. 807–814).

Narkhede, S. (2018). Understanding confusion matrix. Retrieved from https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62

Nwankpa, C., Ijomah, W., Gachagan, A., & Marshall, S. (2018). Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378* (2018).

of California, U. (2018). Why are neuron axons long and spindly? study shows they're optimizing signaling efficiency. Retrieved from https://medicalxpress.com/news/2018-07-neuron-axons-spindly-theyre-optimizing.html

Olabiyi, O., Martinson, E., Chintalapudi, V., & Guo, R. (2017). Driver Action Prediction Using Deep (Bidirectional) Recurrent Neural Network (2017). doi:10.1162/jocn_a_00970. arXiv: 1706.02257

Oliphant, T. E. (2006). *A guide to numpy*. Trelgol Publishing USA.

Ozertem, U., Chapelle, O., Donmez, P., & Velipasaoglu, E. (2012). Learning to suggest: A machine learning framework for ranking query suggestions. In *Proceedings of the 35th international acm sigir conference on research and development in information retrieval* (pp. 25–34). ACM.

Pascanu, R., Mikolov, T., & Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *International conference on machine learning* (pp. 1310–1318).

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* **12** (2011), 2825–2830.

Pines Salomon. (2018). Top 25 causes of car accidents. Retrieved from https://seriousaccidents.com/legal-advice/top-causes-of-car-accidents/

Quiza, R. & Davim, J. (2011). Computational methods and optimization. (pp. 177–208). doi:10.1007/978-1-84996-450-0

Rojas, R. (2013). *Neural networks: A systematic introduction*. Springer Science & Business Media.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review* **65**.(6) (1958), 386.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. Retrieved from http://ruder.io/optimizing-gradient-descent/index.html#gradientdescentvariants

Russel, S. & Norvig, P. (2010). Artificial intelligence: A modern approach third edition. *Person Education, Boston Munich* (2010).

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.* **3**.(3) (1959, July), 210–229. doi:10.1147/rd.33.0210

Scikit Learn. (2019). 3.1. cross-validation: Evaluating estimator performance. Retrieved from https://scikit-learn.org/stable/modules/cross_validation.html

Sen, B., Smith, J. D., Najm, W. G., et al. (2003). *Analysis of lane change crashes*. United States. National Highway Traffic Safety Administration.

Sharma, S. (2017). What the hell is perceptron? the fundamentals of neural networks. Retrieved from https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53

Shung, K. P. (2018). Accuracy, precision, recall or f1? Retrieved from https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

Simonyan, K. & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

Snyman, J. A. (2005). *Practical mathematical optimization*. Springer.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* **15**.(1) (2014), 1929–1958.

Tieleman, T. & Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* **4**.(2) (2012), 26–31.

Tomar, R. S., Verma, S., & Tomar, G. S. (2010). Prediction of lane change trajectories through neural network. *Proceedings - 2010 International Conference on Computational Intelligence and Communication Networks, CICN 2010* (2010), 249–253. doi:10.1109/CICN.2010.59

Van Rossum, G. & Drake Jr, F. L. (1995). *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands.

Volvo Accident Research Team. (2011). European Accident Research Safety Report. *Engineering* (2011), 1–31. doi:ER-624264

Wahde, M. (2008). *Biologically inspired optimization methods: An introduction*. WIT press.

Walia, A. S. (2017). Types of optimization algorithms used in neural networks and ways to optimize gradient descent. Retrieved from https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f

Ward Law Firm. (2018). Changing lanes in an unsafe manner can lead to problems. Retrieved from https://www.855dolor55.com/en/blog/changing-lanes-in-an-unsafe-manner-can-lead-to-problems/

Wikipedia. (2017). Recurrent neural network. Retrieved from https://en.wikipedia.org/wiki/Recurrent_neural_network

Wikipedia. (2018). Long short-term memory. Retrieved from https://en.wikipedia.org/wiki/Long_short-term_memory

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., ... Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

Yung, J. (2016). Explaining tensorflow code for a multilayer perceptron. Retrieved from https://www.jessicayung.com/explaining-tensorflow-code-for-a-multilayer-perceptron/

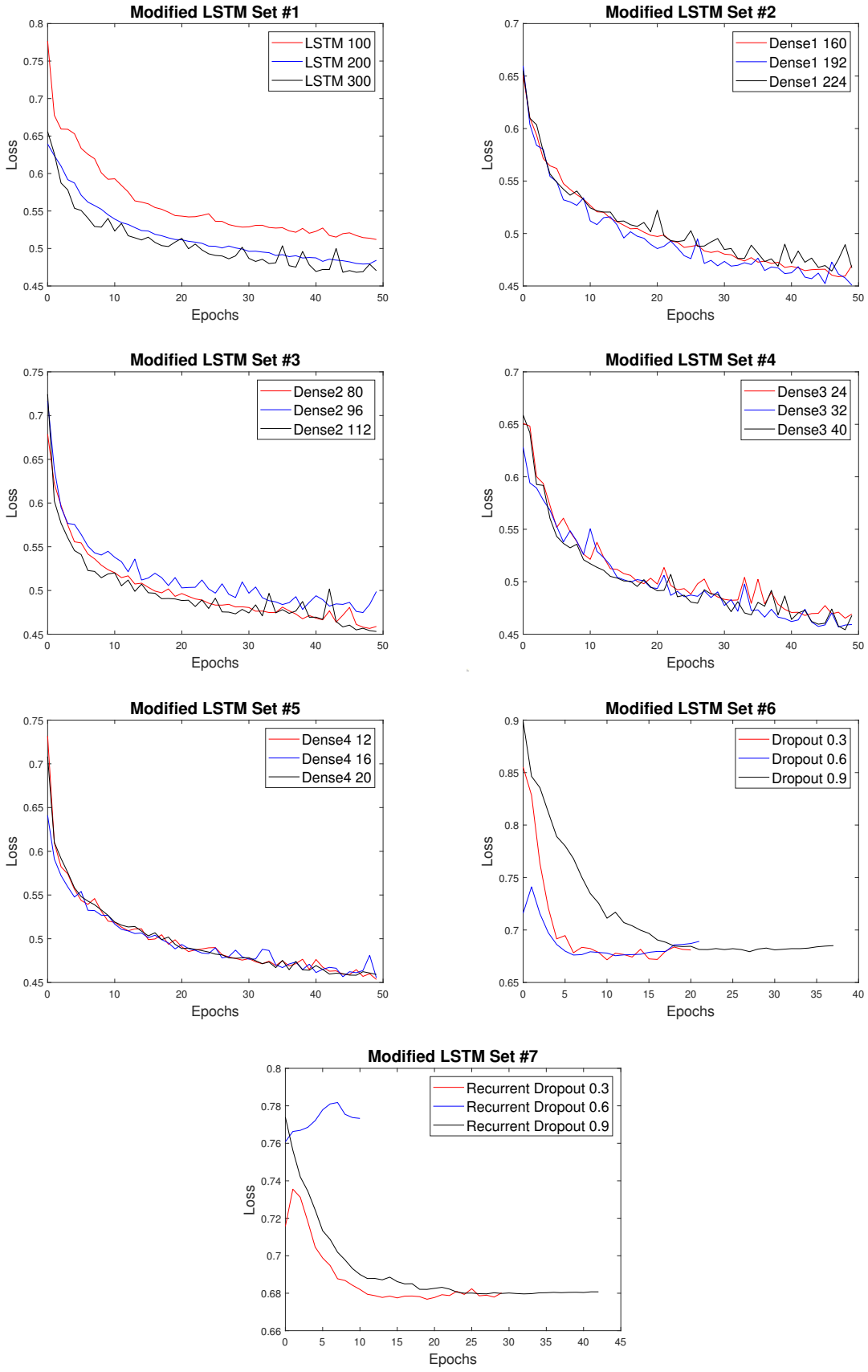

Figure 4.0.7: *Results of hyperparameter tuning conducted on the modified LSTM model*

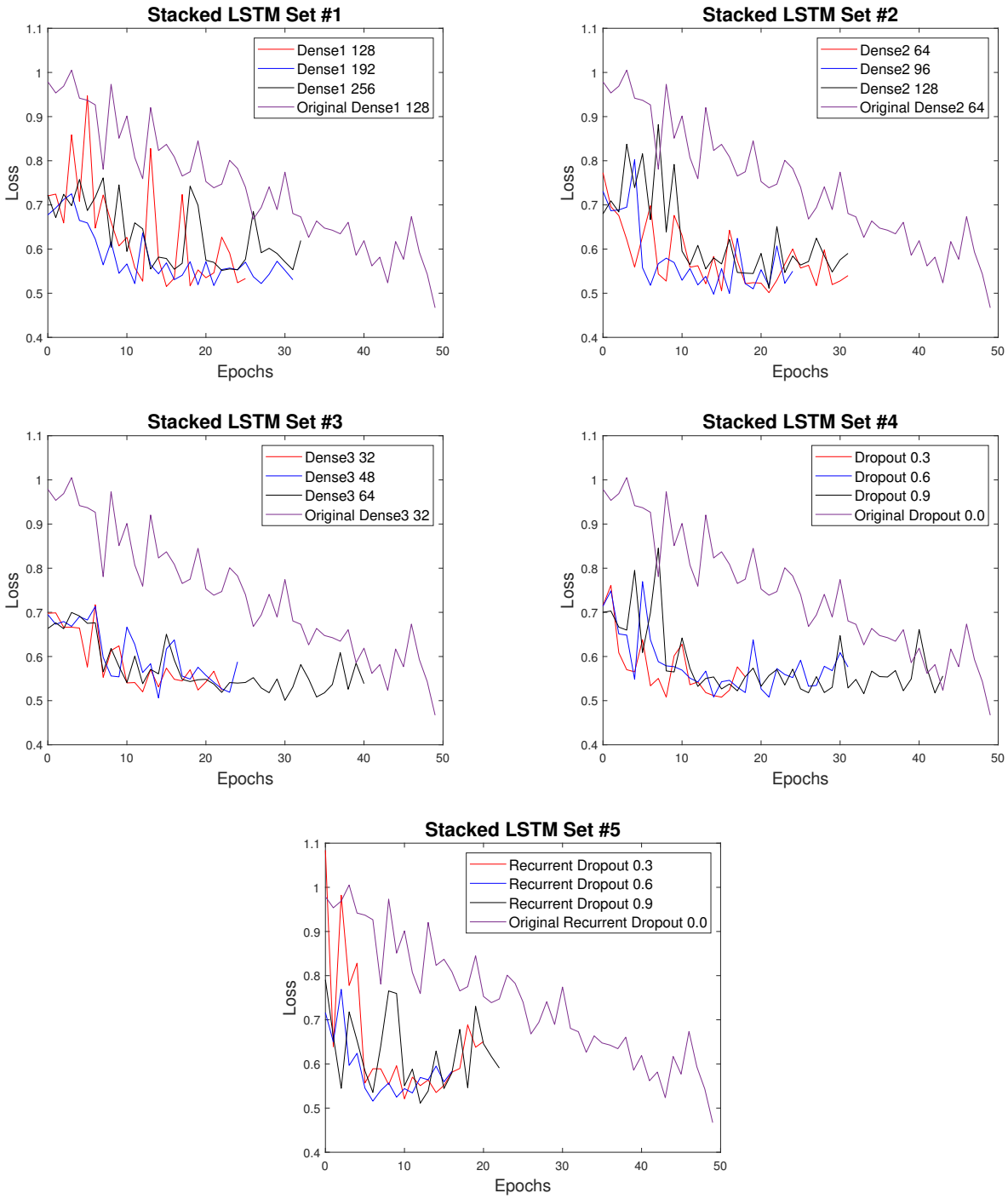

Figure 4.0.8: *Results of the hyperparameter tuning conducted on the stacked LSTM model*