A neural network approach in optimising airport strategy with trajectory prediction.

Bart Rozendaal

(2636955)

30 June 2022

Thesis MSc EOR - Operations Research

Thesis committee:

Prof. dr. Sitters (supervisor)

Drs. F. Dijkstra (company supervisor) Dr. yyyy (co-reader)

1 Introduction

Due to the Covid-19 pandamic, the aviation sector experienced the steepest decline in air traffic demand in history. According to EUROCONTROL (2022a), a decline in European air traffic demand was observable from 11,1 million flights in 2019, to 5,0 million flights in 2020, a decrease of almost 55%. However, in their 2021-2027 forecast (See Figure 1), it is expected that European air traffic is recovered to the levels of 2019 by the end of 2023. Moreover, EUROCONTROL expects that by 2050, European air traffic demand has increased to 16.0 million flights, a growth of 44% compared to 2019.

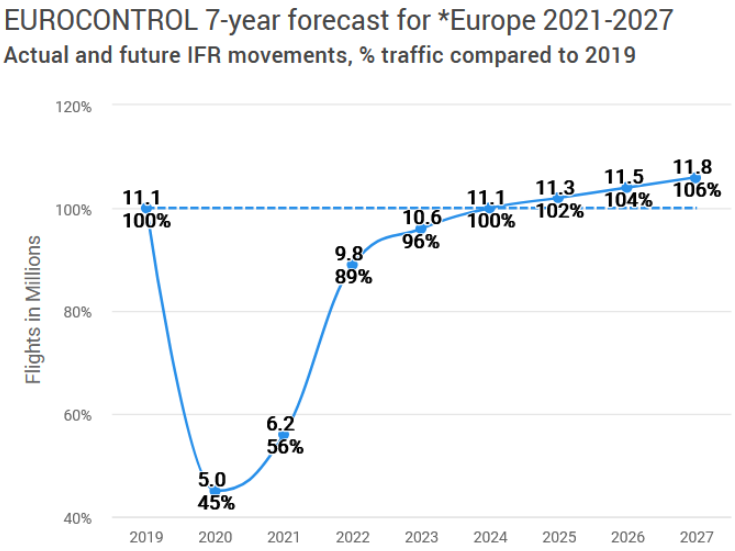

Figure 1: 7-year forecast of European air traffic demand by EUROCONTROL (2022a). (*Europe = European Civil Aviation Conference (ECAC) area)

In order to handle this increase in air traffic, the efficiency of air traffic management (ATM) will have to increase and the performance of the operating systems must improve. The limit on capacity in ATM is mainly caused by human factors such as the cognitive skills and mental workload of the air traffic controllers, who are in charge of directing all the air traffic in an efficient and safe manner. To support the air traffic controllers and increase the airspace capacity, a Decision Support Tool (DST) is developed. How well the DST can help the controller, depends on the quality of the predictions of the aircraft trajectories. Trajectory prediction of an aircraft is the estimation of the future location of an aircraft based on input data such as environmental conditions (weather data), technical information (performance of an aircraft), human interference (actions taken by the pilot or air traffic controller clearances) or current flight information (live location of the flight). In the literature, a distinction is made between short-term and long-term trajectory predictions, where

- Short-term trajectory predictions are based on real-time data of a flight, with the aim to predict the location of an flying aircraft within a few minutes.
- Long-term trajectory predictions are based on historical flight data to predict the entire trajectory of a flight before take off.

For a DST, it is not unusual to take the filed flight plan (FPL) as input, which is a fixed plan determined before take off with all the necessary information about a flight, including the exact route which has to be flown. When the FPL is given as input, the DST often suffers from a lack of information, since in reality, aircrafts often deviate from the route described in their FPL. Smaller deviations are most of the time caused by congested airspace or weather conditions, while bigger deviations often occur when permission is given to cross military airspace to save time or fuel (Rodriguez, 2022). Figure 2 Illustrates a rather large deviation from the FPL. All these factors that affect the trajectory are often not included in the DST and therefore the quality of the tool declines, missing the opportunity to increase capacity.

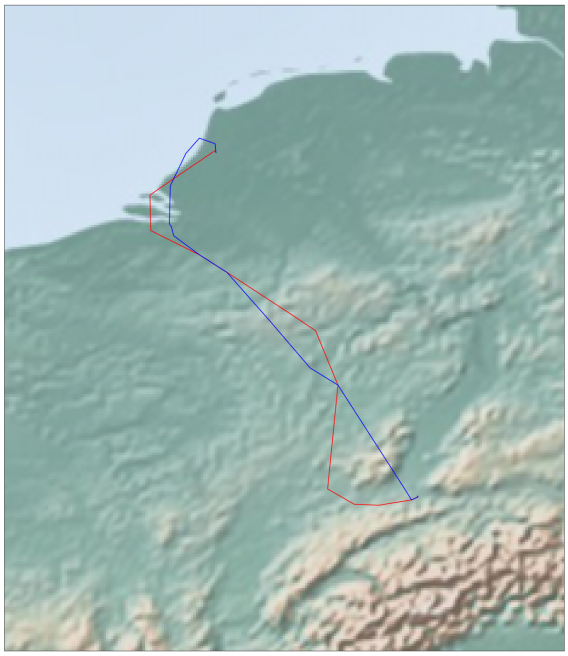

Figure 2: Flight from EuroAirport Basel-Mulhouse-Freiburg to Amsterdam Schiphol Airport where the actual flown route (blue) deviates from the FPL (red).

Currently, the combination of the input data and the models that are used is failing to make accurate trajectory predictions, leading to a deteriorated performance in ATM. The International Civil Aviation Organization (ICAO) regards trajectory based operations as the core of the next generation air navigation systems. The demand for intelligent ATM tools is increasing to make more precise trajectory prediction, leading to better insights when a flight will enter a certain flight zone or when it will arrive at the destination airport. This greatly improves capacity management and increases the air flow that the air traffic control (ATC) can handle. Two main uncertainties are currently seen as highly problematic for good capacity management, which are:

- Delays when departing, leading to uncertain departure times
- Flights not following their FPL, due to air traffic controllers giving other clearances to the pilots.

This paper will propose a Recurrent Neural Network (RNN) to address the second uncertainty described above. A deep Long Short-Term Memory (LSTM) network is developed to solve a

sequence-to-sequence learning problem and predict the most likely route flown by the aircraft before take off, which is more accurate than the route described in the FPL. The RNN with LSTM cells manage to capture the long term dependencies in the data set, predicting a long term trajectory existing out of 2-D way points.

Although other papers have made similar attempts to tackle this problem with , this paper is the first to train the network on historical trajectories only, without need for other input such as aircraft performance, weather data or restricted area information. Furthermore, this paper is the first to apply this idea on a single airport (Amsterdam Schiphol airport, EHAM) for all the incoming flights within the European Civil Aviation Conference (ECAC) area. The data originates from the EUROCONTROL R&D data archive (EUROCONTROL, 2022b). Hyper-parameters such as the hidden units, hidden layers and batch size will be tuned to find the right configuration of the network to make optimal trajectory predictions. Also, additional structures such as a bidirectional network and an encoder-decoder model are applied to boost performance.

(TO DO: give short overview of what's next in the paper)

1.1 Literature Review

As Artificial Intelligence (AI) is developing rapidly, RNN's have become the state of the art for sequence modeling problems such as speech recognition (Graves and Jaitly, 2014), image recognition (Donahue et al., 2014), financial market predictions (Eapen et al., 2019), and natural language processing (Klosowski, 2018). Motivated by these successes, RNN's has also been applied to trajectory predictions for vehicles (Lv et al., 2017), as well as for aircrafts. Unlike traditional neural networks, RNN's can operate with LSTM cells, which have a memory function that helps to capture relations in time series and other sequential data, hence the reason why LSTM is applied so often in trajectory prediction (Hochreiter and Schmidhuber, 1997).

Multiple machine learning attempts have been made to improve airspace capacity, also non-neural network approaches. Salaun et al. (2012) used clustering techniques to develop three types of proximity maps to support air traffic controllers: a presence map which indicates the local density of traffic, a conflict map that highlights locations with potential conflicts and a outlier proximity map which show probabilities of proximity's between flights belonging and flights not belonging (so called 'outliers') to dominant flows of traffic. Included was a traffic flow manager with information relating to the complexity and difficulty of managing an selected airspace. The purpose of the maps is to anticipate how outliers impact the dominant traffic flow and how traffic flows interact with each other. Murça et al. (2016) contributed to post-event analyses in terms of performances and inefficiencies. They provided a guidance for real time decision support with a data mining framework for air traffic flows based on radar data. A density-based clustering algorithm was used which was able to identify major flight trajectory patterns and detect non-conforming behavior.

Ayhan and Samet (2016) used a Hidden Markov Model that was mainly focused on the environmental data to make trajectory predictions based on historical trajectories. In their model, they considered the airspace as a 3-D grid network, where each grid point is a location with a weather

observation. Hypothetical cubes were build around these grid points, such that the whole airspace was considered as a set of cubes. Every cube contained its own set of weather parameters such that, when predicting a trajectory, one could accurately take into account the environmental (weather) conditions at different points in the track. However, the major drawback of this approach is that only the weather conditions closest to the current point in the track are considered, while ATC usually takes into account the weather conditions in a larger region. Also, the paper was only focused on one flight, DAL2173 from ATL to MIA. Applying this approach to multiple flights in a bigger airspace, would take a tremendous amount of extra memory and computing power.

Shi et al. (2007) were, to the knowledge of the authors, the first to introduce a LSTM network for flight trajectory prediction, which can process long-term sequences. Because dynamic features in the data such as acceleration and speed were weakened or averaged in the data, a sliding window technique was used to maintain continuity of the sequence. This technique predicts the trajectory by reading the data step by step, instead of working with average values from the data. The trajectory was predicted using intermediate fases, which were achieved by the sliding window technique, improving the accuracy. Shi succeeded to outperform Markov Models, which were at the time the most accurate models to predict aircraft trajectories. Multiple papers (Liu and Hansen (2018), Ma and Tian (2020) and Pang et al. (2019)) combined the idea of approaching the airspace as a set of cubes (Ayhan and Samet, 2016) and using a neural network to perform aircraft trajectory predictions. Convolutional RNN's with an encoder-decoder structure were developed, where the encoder layer takes historical flight plans as input to learn the temporal relations in the data. the convolutional layer, which is often used for image recognition, takes the meteorological data-set (air temperature, wind speed, etc..) as input to learn the the spatial aspects for the prediction. Both the encoder and convolutional layer give their information to the decoder layer which translates the information and produces the actual prediction. However, for all papers it holds that the predictions were made for single flights (from Houston to Boston (Liu and Hansen, 2018), from Qingdao to Beijing (Ma and Tian, 2020) and from New York to Los Angeles (Pang et al., 2019)).

Naessens et al. (2017) developed a more comprehensive deep neural network that was trained on more predictors such as historical trajectories, reservation of military areas, day of the week, and the destination airport. The network was used to predict the most likely route through the Maastricht UAC airspace and was substantially more accurate than other the models used in that time, although more data intensive. Recently, Jia et al. (2022) proposed a RNN with an attention mechanism, which improves the predictions by learning to what parts in the sequential data the network should pay more 'attention' to. Especially in longer sequential data, the attention mechanism showed great potential by strengthen the influence of the most import data, and attenuate the influence of unnecessary factors, which improves the prediction accuracy.

Besides predicting trajectories for improving airspace capacity, Zhang and Mahadevan (2020) proposed a Bayesian neural network for flight trajectory prediction with the aim for safety assessment. In this approach, two types of deep learning models are trained. One deep feedforward neural network (DNN) to make one-step-ahead predictions, and in parallel, a deep LSTM network is trained to make long term predictions. The DNN appeared to be more accurate, but obviously has the downside of the short prediction horizon. Therefore, a multi-fidelity prediction was created by blending the 2 models together, where a long-term prediction was made by the LSTM

network, which was corrected by the DNN as the flight was progressing, leading to very accurate predictions. The approach was applied to multiple flights to assess safety by means of separation. The high precision made the approach very suitable for safety assessment purposes, however, for improving air space capacity it was less suitable, considering that the high precision was achieved at a later stage in the prediction by the DNN. Zhengfeng et al. (2021) developed a Social Long Short-Term Memory (S-LSTM) which was able to consider the interaction between aircrafts and established a multi-aircraft collaborative prediction model. A pooling layer was created to share the information of all the different flights, which could effectively capture the interactive information between flights. Although safety assessment was the aim of the network, this approach could be promising for capacity management as well.

Less studies have been conducted to the actual cause of deviations of flight plans. Bongiorno et al. (2016) brought understanding in the relation between realized and planned flights and the determinants of flight deviations. It was found that realized trajectories were shorter on average than planned trajectories, especially during night-time. Also, it was concluded that deviations occur more often close to the departure airport and during low traffic phases. This evidence suggests that most deviations are caused by air traffic controllers clearances, when the traffic conditions allow this. Besides these findings, a new metric, called di-fork, was introduced to characterize navigation points according to the likelihood that a deviation would occur there.

1.2 Research Question

Considering the problem formulation and the reviewed literature above, this paper will propose a model to improve airspace capacity by long-term aircraft trajectory prediction. While achieving this, a model with minimal data and computing power is desirable, such that it is actually suitable to apply it on any selected airport, for a variety of (if not all) incoming flights. The research done is this paper to develop such a model has answered the following research question:

'What are optimal settings for a recurrent neural network for making aircraft trajectory predictions?'

To give a complete and precise answer to the research question, the following sub question were considered:

- *What is the optimal number of hidden layers, and neurons per layer for the RNN?*
- *What is the best optimisation algorithm, and what is the optimal batch size for the RNN?*
- *Can the performance of the RNN be improved by the use of structures such as a bidirectional network or an encoder-decoder model?*

2 Neural Networks

2.1 Feed-forward Neural Network

A neural network (NN) refers to a type of self-learning algorithms that is able to detect relations in a data set. As the name already suggests, the algorithms way of learning is inspired by the human nervous system. In short, the human nervous system exists of cells called neurons. Neurons are connected with each such that a given neuron can receive signals from a set of other neurons, let's say the 'input', and then based on this, send signals to the next set of neurons, providing an 'output'. NNs are designed to work in a similar way. Every NN consists of a input layer, an output layer and one or several 'hidden' layers (a NN with more than one hidden layer is referred to as a deep NN). Focusing on one layer, an arbitrary number of neurons can be found. A neuron in layer i in never connected to another neuron in the layer i, but is connected to all other neurons in layers $i-1$ and $i+1$ (except from the input and output layers, where a neuron is strictly connected to all neurons in layers $i+1$ and $i-1$, respectively), see figure 3.

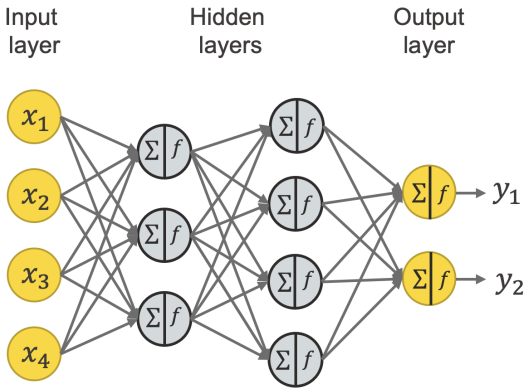

Figure 3: a systematic representation of a neural network, from Melcher (2021).

The input of a NN is an numerical data set, such that all the neurons in the input layer together can present this data by holding a certain value. Inside the others neurons in the other layers, a simple operation in performed. First, a neuron adds up all the input $(x_1, x_2, ..., x_n)$ that it receives from the neurons in the previous layer, multiplied by its own arbitrary weights $(w_1, w_2, ..., w_n)$ such that we have the following:

$$z = \sum_{i=1}^{n} x_i w_i + b.$$

Where b is an arbitrary bias, which will be explained later. Then, an activation function, f, is used to limit z between 2 values, mostly between 0 and 1. Which activation function is used, depends on the type of problem, but probably the most common activation function is the sigmoid function, which is defined as:

$$\sigma(x) = \frac{1}{1+e^{-x}}.$$

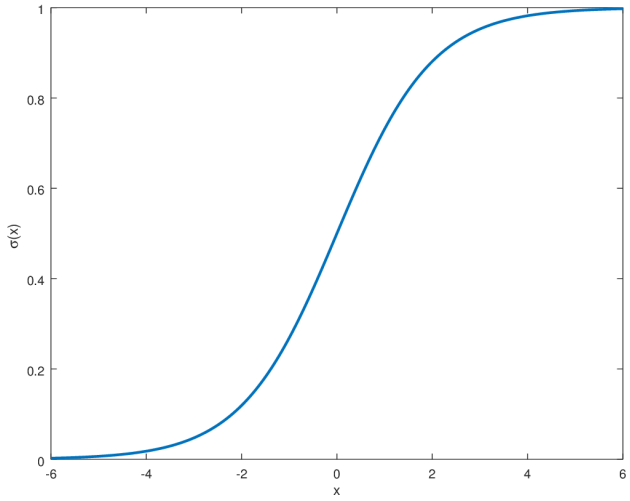

Figure 4: The sigmoid function plotted over interval [-6,6], from Berg (2019).

Finding that the final output, or 'activation', of a neuron can be defined as:

$$a = f(\sum_{i=1}^{n} x_i w_i + b). \tag{1}$$

Where f is the activation function (identical for all neurons within the same layer), $x_1, x_2, ..., x_n$ the input (identical for all neurons within the same layer), $w_1, w_2, ..., w_n$ the neuron specific weights and b the neuron specific bias. Looking at 3, the intuitive meaning of the bias can be understood. The biases is able to shift the linear combination of the inputs, determining whether a given neuron will send a message to the next neurons or not. Choosing b very negative will result in a sigmoid function being very close to 0, (almost) silencing the neuron.

Every neuron in every layer (except from the input layer) will perform the operation above, such that eventually the output layer will provide the final output. When all neurons in the network performed such an operation once, a forward pass, or forward-propagation, is completed. The network described above is the most basic form of a NN and is called a feed-forward neural network (FFNN). After the forward-propagation, the back-propagation is performed, which is explained later. As said, the weights and biases in the neurons are chosen arbitrarily, meaning that the initial output is just a random transformation of the input. Besides a given input, a NN will also require an given output. This given output is compared to the output from the output layer. If the two outputs correspond well with each other, then the current weights and biases in the network are well adjusted, if not, the weights and biases need fine-tuning. To determine how well the output of the network corresponds to the given output, a cost function is used. As with the activation function, the cost function depends on the type of problem. A common cost function is the mean squared error (MSE), defined as:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2.$$

Where Y_i is the true value and $\hat{Y}_i$ the predicted value. The more accurate the predictions, the lower the cost function. When a NN is learning the relations in a data set, and thus making more accurate

predictions, actually comes down on minimising the cost function. Minimising the cost function is done via a process called back-propagation.

2.1.1 Back-propagation

Back-propagation is the algorithm used to obtain all partial derivatives of the cost function w.r.t. the weights and biases, such that the cost function can be minimised. All the partial derivatives together form the gradient of the cost function. The gradient of a function captures the change in output of a function, w.r.t. the change in one of the inputs of the function, meaning that the gradient gives the NN the information needed to change the weights and biases to minimise the cost function. Since the final output depends on multiple operations, each executed in a different neuron, the chain rule is required to calculate the gradient. For simplicity, consider a network with an input layer, one hidden layer and an output layer, all existing of one neuron. An example is provided how the weight in the output layer is adjusted to minimize the cost function. Later, the explanation will be extended to a more complicated network.

denote the activation of the neuron in the output layer as $a^{(L)}$, where L serves as a subscript to the output layer. The desired output is denoted as y, such that the cost function for a single training example can be defined as:

$$C_0(...) = (a^{(L)} - y)^2.$$

Now, $a^{(L)}$ can be written as:

$$a^{(L)} = \sigma(z^{(L)}) \text{ where } z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)}.$$

To determine how to adjust $w^{(L)}$ to minimize the cost function, $\frac{\partial C_0}{\partial w^{(L)}}$ needs to be determined using the chain rule:

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial C_0}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w^{(L)}}$$

where

$$\frac{\partial C_0}{\partial a^{(L)}} = 2(a^{(L)} - y)$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

$$\frac{\partial z^{(L)}}{\partial a^{(L)}} = a^{(L-1)}.$$

Finally, the average of multiple training samples (called a batch) is taken to determine $\frac{\partial C}{\partial w^{(L)}}$:

$$\frac{\partial C}{\partial w^{(L)}} = \frac{1}{m} \sum_{k=0}^{m-1} \frac{\partial C_k}{\partial w^{(L)}}.$$

This expression can now be used to update $w^{(L)}$ using a learning rate ϵ, which determines the influence of the derivative on the weight:

$$w^{(L)} = w^{(L)} - \epsilon \frac{\partial C}{\partial w^{(L)}}.$$

Notice that the weight is updated in the direction of the steepest descent of the cost function. the learning rate determines the size of the step to the optimum value i.e. the minimum value of the cost function. A greater learning rate will speed up the learning process of the NN, but increases the risk of overshooting the optimal value. Updating all the weights and biases according to the algorithm above is called stochastic gradient descent and will be discussed in more detail later in the paper. Minimising the loss function is done w.r.t. all the weights and biases in the NN, meaning that the derivative above forms only one of the elements of the entire gradient of the loss function:

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \end{bmatrix}.$$

Extending this example to a more complicated network with let's say, n neurons per layer, the method stays unchanged, only the notation needs to be adjusted. The activation of the j'th neuron in the output layer is now denoted as $a_j^{(L)}$, resulting in a loss function of:

$$C_0 = \sum_{i=1}^{n_L} (a_i^{(L)} - y_i)^2.$$

To be able to make a distinction between all the weights, the weight in neuron j of layer L (the output layer in this case), multiplying activation $a_k^{(L-1)}$ is denoted as $w_{jk}^{(L)}$. The weighted sum of the activations and the weights then changes into:

$$z_j^{(L)} = w_{j0}^{(L)} a_0^{(L-1)} + w_{j1}^{(L)} a_1^{(L-1)} + w_{j2}^{(L)} a_2^{(L-1)} + ... + w_{jn}^{(L)} a_n^{(L-1)} + b_j^{(L)}$$

where the activation is defined as:

$$a_j^{(L)} = \sigma(z_j^{(L)}).$$

Coming finally to the the derivative:

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial C_0}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}}.$$

Again, the above expression is determined for a whole batch and the average is taken to update the weight using a learning rate. The derivatives of all weights and biases together from the gradient of the loss function. This gradient tells the NN how to update the weights and biases to minimize the cost function, which translates to the NN learning the relations in the data set. Learning these relations is done with the training set, which roughly makes up 70% to 80% of the full data set. The remaining data is called the test set and is used to actually measure the performance of the network. The idea is that the network should not be judged on data that it has already seen while training. When the test set is given to the NN, the network makes predictions by transforming the data with the weights and biases that are achieved during the learning process. The predictions are then compared to the true outcomes and, based on the type of problem, the performance is determined.

2.2 Recurrent Neural Network

FFNNs have shown great potential with data points that are independent of each other. However, when sequential or time-dependent data is used, FFNNs fail to capture the relations between the data points. This is because information always flows in one direction, the forward pass. Information never touches a neuron twice and the output of the model is never re-used as new input. A FFNN can only consider its current input and has no memory, causing that it is unable to detect relations over time. A NN with loops or feedback connections that enables the output to be re-used as input is called a Recurrent Neural Network (RNN). RNNs are now the state of the art when identifying relations in sequential or time-dependent data. RNNs consider their own output from previously seen data as new input when evaluating new data, creating an internal memory. The internal memory is indispensable when identifying relations between depend data points. Figure 5 visualizes the information flow through both networks.

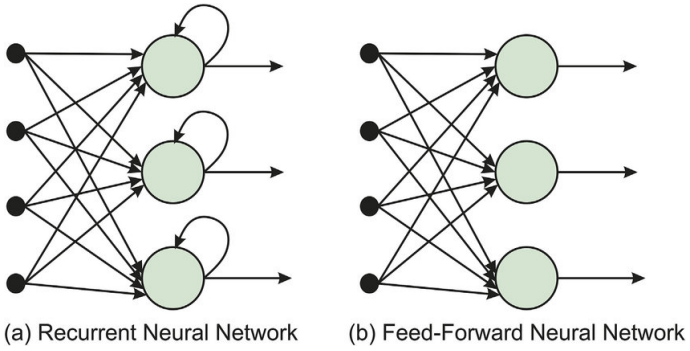

Figure 5: Comparison between a RNN and a FFNN, where information in a FFNN flows in one direction, while information in a RNN returns in loops. from Eliasy and Przychodzen (2020).

For example, given is an algorithm that is supposed to make words suggestions to the user while typing the phrase 'recurrent neural network'. When the user finished the word 'neural', a FFNN cannot predict what comes next since it only considers the word 'neural'. A RNN considers both the words 'recurrent', which is returned by a loop, and the word 'neural', which is given as current input. The RNN is able to predict that 'network' comes next in the sentence.

Creating the internal memory for the RNN caused the architecture of the network to change. A RNN exists of multiple cells which could be described as multiple FFNNs, all taking a single time step from the data, x_t, and a 'hidden state' from the previous cell, s_{t-1} (the first cell takes 0 as hidden state). The hidden state gives the RNN its memory by providing a new cell with information from the previous cell (see figure 6). x_t is transformed by a set of weights, U. For simplicity, the bias vectors are left out, although these do appear in the same way as in FFNN. s_{t-1} is transformed by another set of weights, W. The transformed variables are used as input for an activation function such that the new hidden state s_t is found (equation 2). The weights V are used to transform s_t and using a second activation function the output of the cell, o_t, is attained (equation 3). Based on the type of problem, o_t is passed to an output layer which gives the final output. s_t is passed to the new cell again. An unfolded RNN is shown in figure 6.

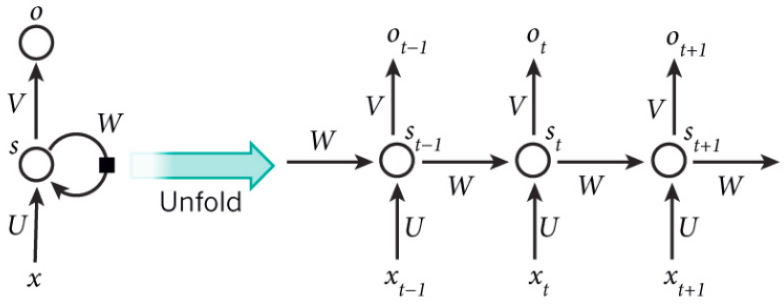

Figure 6: An unfolded RNN existing of multiple cells. Every cell takes a hidden state s_{t-1} and an input x_t, has a set of weights W, U and V and provides an ouput o_t and a new hidden state s_t. From floyd (2019).

2.2.1 Back-propagation through time

As the architecture of a RNN changed, the back-propagation algorithm used for FFNN cannot be applied. When RNNs are trained, a adjusted algorithm called back-propagation through time is used. The main issue that is solved by this algorithm is that the network is not independently trained at a time t, but as well for all time steps before time t. Considering figure 6, the hidden state s_t and output o_t are defined as:

$$s_t = f_1(Ux_t + Ws_{t-1}) \tag{2}$$

$$o_t = f_2(Vs_t) \tag{3}$$

where f_1 and f_2 are activation functions. When performing back-propagation through time on time step t_2, the loss function is defined as:

$$C_2(o_2, y_2) = (o_2 - y_2)^2$$

where y_2 is the true value at $t = 2$. If the weights transforming the hidden states, W, need to be adjusted, the derivative of the loss function w.r.t. these weights need to be determined:

$$\frac{\partial C_2}{\partial W} = \frac{\partial C_2}{\partial o_2}\frac{\partial o_2}{\partial s_2}\frac{\partial s_2}{\partial W} + \frac{\partial C_2}{\partial o_2}\frac{\partial o_2}{\partial s_2}\frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial W}. \tag{4}$$

Notice that when taking the derivative w.r.t. W, the partial derivative w.r.t. all time steps should be included since all hidden states in the past are a functions of these weights. Therefore, the general formula can be described as:

$$\frac{\partial C_N}{\partial W} = \sum_{t=1}^{N} \frac{\partial C_N}{\partial o_N}\frac{\partial o_N}{\partial s_t}\frac{\partial s_t}{\partial W}. \tag{5}$$

A major problem of back propagation through time is a problem called the vanishing (or exploding) gradient problem. Focusing on the term $\frac{\partial s_2}{\partial s_1}$ in equation 4, Notice that

$$s_t = f_1(Ux_t + Ws_{t-1}),$$

meaning that

$$\frac{\partial s_t}{\partial s_{t-1}} = f_1'(Ux_t + Ws_{t-1}) \cdot W. \tag{6}$$

Combining equations 5 and 6:

$$\frac{\partial C_N}{\partial W} = \sum_{t=1}^{N} \frac{\partial C_N}{\partial o_N} \frac{\partial o_N}{\partial s_N} \Big(\prod_{k=t+1}^{N} f_1'(U x_k + W s_{k-1}) \cdot W \Big) \frac{\partial s_t}{\partial W}. \tag{7}$$

As N gets large (generally larger than 10), the first terms of equation 7 tend to vanish to 0 as the derivative of the activation function, in this case most often the tanh function, is always smaller than or equal to 1 i.e.

$$\max_{x \in \mathbb{R}} f'(x) = 1.$$

The exploding gradient problem occurs when W is big enough to overpower the small outcome of $f'(x)$. When the gradient vanishes, the earlier hidden states loose their effect in the whole gradient, meaning no long term dependencies are learned. When the gradient (of some hidden states) explodes, the entire gradient blows up such that computation become unstable. In both cases, the information from the gradient is poor due to lost information (vanishing gradient) or overmagnified information (exploding gradient).

2.2.2 Long Short-term Memory (LSTM)

A solution to the problem is found by using Long Short-Memory (LSTM) cells. LSTM cells have a special structure existing of 4 layers, divided over 3 gates: The forget gate, the input gate and the output gate. Together, the gates maintain two states that are responsible for the memory of the network; the hidden state, known from regular, or 'vanilla', RNNs, and the cell state, typical for LSTM RNNs only. The hidden state has the task to provide the cell with information about the most recent time steps, as in vanilla RNNs. While the cell state serves as a global or aggregate memory of all time steps for the cell, which is missing in vanilla RNNs. The cell state runs as a horizontal line through the cells, see figure 7, while only minor changes are made through linear interactions in the gates.

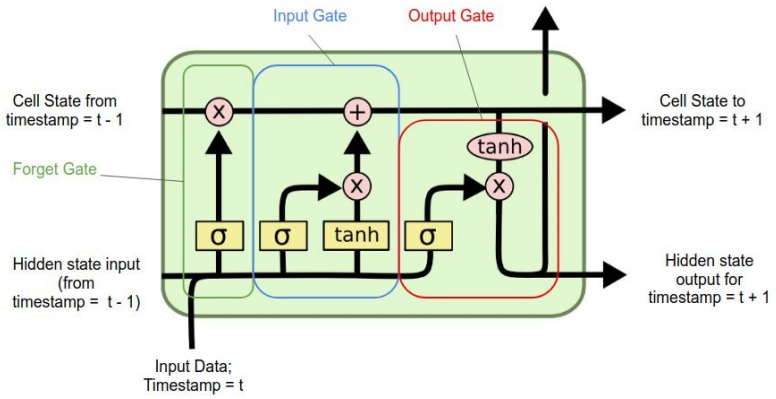

Figure 7: A LSTM cell existing of a forget gate (1 layer), input gate (2 layers) and an output gate (1 layer), producing a new cell state and hidden state. From J. (2020).

- **Forget Gate** The first layer in the cell is located in the forget gate. This layer uses its weights, W_f, to transform the hidden state from the previous cell, h_{t-1}, and input from the data, x_t, and uses the sigmoid function to output a value between 0 and 1. The activation of

the layer is multiplied with the previous cell state c_{t-1}. If the activation is close to 1, most information in c_{t-1} will be saved, while an activation close to 0 means that most information in c_{t-1} will be deleted

- **Input Gate** The second and third layer in the cell are found in the input gate. Both layers consider the same input as the layer in the forget gate. The second layer again contains a sigmoid function, that outputs a value between 0 and 1 to decide to what extent new information should be added. The third layer uses the tanh function as activation functions which outputs a value between -1 and 1, deciding what information is added to c_{t-1}. The activations of both layers are multiplied such that this new information is then added to c_{t-1}, forming the new cell state c_t. Notice that both layers contain their own set of weights, W_{i1} and W_{i2}, to transform the input before it is used as actual input for the activation functions.
- **Output Gate** The fourth layer is in the output gate and determines the actual output of the cell o_t. This is done with a sigmoid function again and using the same input (transformed by the weights of the layer) as the previous layers. the output o_t is multiplied with a filtered version of the new cell state c_t, using the tanh function, which gives the new hidden state of the cell h_t. h_t is used as the output of the cell and is passed to the output layer as in the vanilla RNN. At the end of the cell, both h_t and c_t are passed to the new cell.

The structure of a LSTM cell makes that the gradient will not vanish nor explode if the expression $\prod_{k=t+1}^{N} f_1^{'}(Ux_k + Ws_{k-1})W$ from equation 7 does not vanish nor explode. The cell state of the LSTM cell plays the key role in preventing this. Recall that:

$$c_t = \sigma(W_f[x_t, h_{t-1}]) \cdot c_{t-1} + \sigma(W_{i1}[x_t, h_{t-1}]) \cdot tanh(W_{i2}[x_t, h_{t-1}])$$

which will be written as:

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{C}_t.$$

such that the expression is easy to interpret. The new cell state at time t is the sum of the old cell state at time $t-1$ multiplied by an activation between 0 and 1 to indicate how much is saved from the old cell state, and a new 'candidate' cell state multiplied by an activation between 0 and 1 to indicate how much new information is added to the new cell state.

Reconsidering equation 6, but now applied to the cell states:

$$\frac{\partial c_t}{\partial c_{t-1}} = \frac{\partial}{\partial c_{t-1}}(f_t \cdot c_{t-1}) + \frac{\partial}{\partial c_{t-1}}(i_t \cdot \tilde{C}_t) =$$

$$\frac{\partial f_t}{\partial c_{t-1}} \cdot c_{t-1} + f_t + \frac{\partial i_t}{\partial c_{t-1}} \cdot \tilde{C}_t + \frac{\partial \tilde{C}_t}{\partial c_{t-1}} \cdot i_t$$

where

$$\frac{\partial f_t}{\partial c_{t-1}} \cdot c_{t-1} = \sigma^{'}(W_f[x_t, h_{t-1}]) \cdot W_f \cdot \frac{\partial h_{t-1}}{\partial c_{t-1}} \cdot c_{t-1} =$$
$$\sigma^{'}(W_f[x_t, h_{t-1}]) \cdot W_f \cdot o_{t-1} \cdot tanh^{'}(c_{t-1}) \cdot c_{t-1},$$

and

$$\frac{\partial i_t}{\partial c_{t-1}} \cdot \tilde{C}_t = \sigma^{'}(W_{i1}[x_t, h_{t-1}]) \cdot W_{i1} \cdot \frac{\partial h_{t-1}}{\partial c_{t-1}} \cdot \tilde{C}_t =$$
$$\sigma^{'}(W_{i1}[x_t, h_{t-1}]) \cdot W_{i1} \cdot o_{t-1} \cdot tanh^{'}(c_{t-1}) \cdot \tilde{C}_t,$$

and

$$\frac{\partial \tilde{C}_t}{\partial c_{t-1}} \cdot i_t = tanh^{'}(W_{i2}[x_t, h_{t-1}]) \cdot W_{i2} \cdot \frac{\partial h_{t-1}}{\partial c_{t-1}} \cdot i_t =$$
$$tanh^{'}(W_{i2}[x_t, h_{t-1}]) \cdot W_{i2} \cdot o_{t-1} \cdot tanh^{'}(c_{t-1}) \cdot i_t.$$

The fundamental difference between the expressions $\frac{\partial s_t}{\partial s_{t-1}}$ in the vanilla RNN and $\frac{\partial c_t}{\partial c_{t-1}}$ in the LSTM RNN, is that $\frac{\partial c_t}{\partial c_{t-1}}$ exists of a sum of 4 terms, whereas $\frac{\partial s_t}{\partial s_{t-1}}$ is a product of 2 terms. This makes that the LSTM RNN is more capable in balancing the expression. If $\frac{\partial c_t}{\partial c_{t-1}}$ tends to vanish to 0, the second term, f_t, can easily be increased to prevent this. This is not as easy for the vanilla RNN, since $\frac{\partial s_t}{\partial s_{t-1}}$ is a product with at least one term smaller than or equal to 1, instead of a sum. Actually, as f_t is a function of the weights which are adjusted in the learning process, the LSTM RNN can learn when to preserve the gradient, or when to let it vanish.

2.2.3 Bidirectional Long Short-term Memory (BI-LSTM)

LSTM RNNs are complex and advanced networks, especially compared to FFNNs. A LSTM layer exists of multiple cells, with each cell containing multiple layers. Every cell outputs two states which help to keep a memory in the network. As with FFNNs, multiple attempts are made trying to improve the performance of a LSTM RNN by creating a network with multiple layers, making it a deep NN. Adding, or stacking, multiple LSTM layers can indeed improve the performance. The stacked LSTM layers are sometimes able to capture more complex relations in the data, which was also observed for FFNN. However, the fact that LSTM layers can learn relations in sequential data, caused that other architectures were invented. Bidirectional networks are one example is this.

In a bidirectional network, an extra LSTM layer is added to the network. In this layer, information flows in the opposite direction i.e. it takes the sequential data is the reversed order. A BI-LSTM is capable of utilizing the information from both sides, which enables it to model dependencies between data points in the data in both directions. When a BI-LSTM is trained, it combines the information from both LSTM layers to provide an output. The bidirectional structure provides the network with a richer context which showed improved performance for tasks such as natural language processing. The improved performance can be explained by the fact that every output has information about both the past and the future. Figure 8 visualizes a BI-LSTM network.

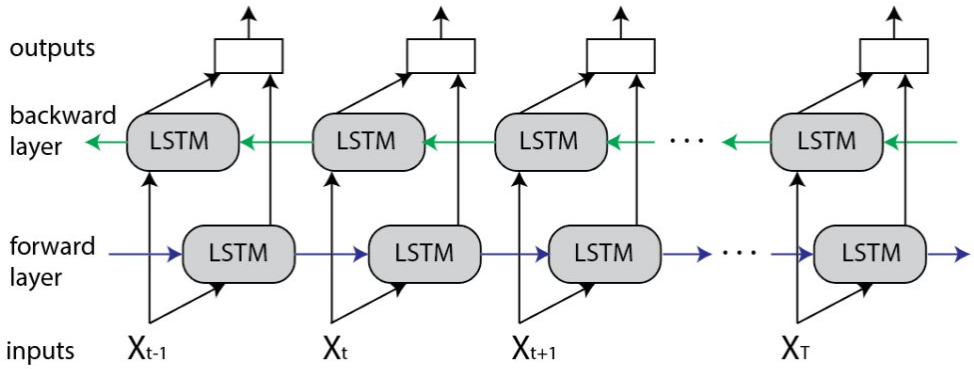

Figure 8: An unfolded BI-LSTM network with two LSTM layers taking the data in the reversed order. From Zvornicanin (2022).

2.2.4 Encoder-Decoder Model

RNNs became popular as the interest in finding relations in sequential data increased. Especially in language related tasks, data was used where the input did not correspond with the output in terms of length. As LSTM networks were designed to take one time step from the data en directly transform this to an output, it was unable to perform well on data were input and output length varied. The solution for this problem was found in the encoder-decoder model. An encoder-decoder model exists of 2 LSTM layer; an encoder layer and a decoder layer. The encoder layer works as usual and takes the input step by step and produces a hidden state and cell state for every step. When the last time steps is processed, the last encoder layer cell provides a hidden state en cell state and 'encodes' the information in a context vector. The context vector is a fixed size vector for every input sequence length and is used by the decoder layer as initial hidden state and cell state. Based on the context vector, the first cell in the decoder layer 'decodes' the context vector and predicts the first time step of the predicted sequence. The decoder is then formed in such a way, that the output from the previous cell is used as input for the next cell. Meaning that based on the last output and the 2 states from the last cell, the next time step in the predicted sequence is predicted. The decoder layer learns when a sequence is completed and stops with predicting. The architecture of the model enables the network to predict sequences that have a different length than the input sequence. A systematic representation of the encoder-decoder model is presented in figure 9.

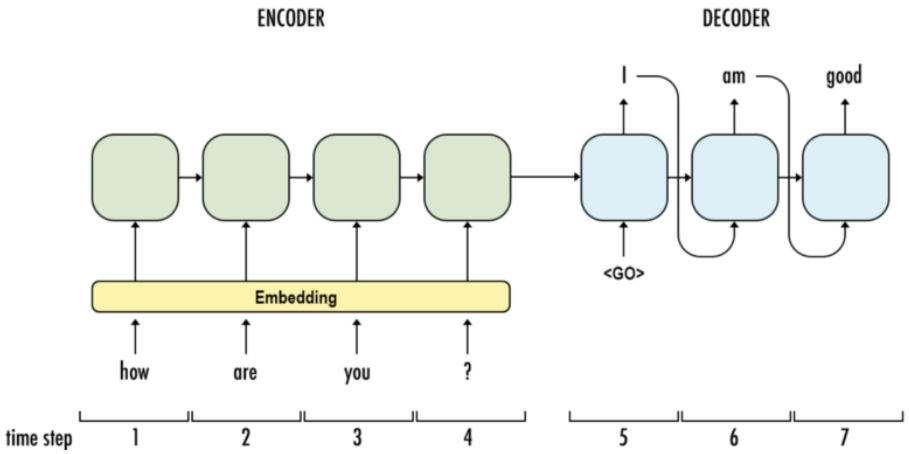

Figure 9: An encoder-decoder model where the encoder provides a context vector for the decoder, which outputs the predicted sequence. From Chablani (2017).

2.3 Optimisation

As mentioned in subsection 2.1.1, stochastic gradient descent (SGD) is the algorithm used to minimize the cost function in NNs. Gradient decent in general refers to an algorithm that updates the parameters, $\theta \in \mathbb{R}^n$, of an objective function, $J(\theta)$, step by step to minimise the function. In order to do that, the gradient of the function w.r.t the parameters is determined, $\nabla_\theta J(\theta)$, after which the parameters are updates in the opposite direction of the gradient. A learning rate ϵ determines the step size, so to what extend the parameters are changed. In principle, gradient descent starts somewhere at random, evaluates the slope at that point and uses the slope to move

towards the minimum of the function:

$$\theta_{t+1} = \theta_t - \epsilon \cdot \nabla_\theta J(\theta_t).$$

Although the algorithm guarantees to converges to a (local) minimum, gradient descent can be time consuming and computationally heavy. This is mainly because it computes the gradient for all training samples and takes an average of this before making a single update. SGD is faster and takes less computing power by computing the gradient for one randomly selected training sample, and then updates θ only using this information. The downside of SGD is that the functions value can fluctuate heavily in the process of finding the minimum, see figure 10.

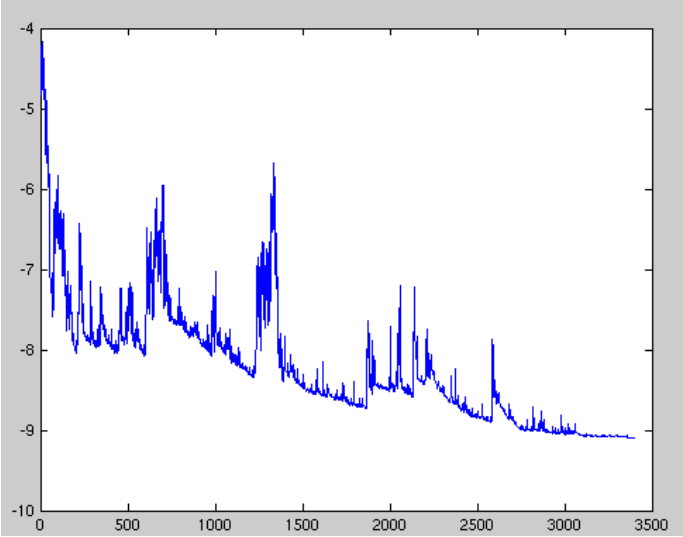

Figure 10: The fluctuation of the objective function when minimising with SGD. From Ruder (2016).

Instead of using one randomly chosen sample to base the parameter updates on, SGD is often executed on a batch of samples, sometimes called mini-batch gradient descent. Mini-batch gradient descent combines the low fluctuation of gradient descent and the speed of SGD. A greater batch leads to lower a fluctuation, but also leads to lower a speed. Another method to decrease fluctuations in SGD is called momentum, where a fraction of the past direction is added to the current direction when updating θ. If the partial derivative of the objective function w.r.t. a given parameter has the same direction in consecutive time steps, the terms strengthen each other and the update in that direction is increased. If the directions go against each other, the terms cancel out and the update in that direction is decreased. The update rule for SGD with momentum is as follows:

$$v_t = \gamma v_{t-1} + \epsilon \cdot \nabla_\theta J(\theta_t)$$

$$\theta_{t+1} = \theta_t - v_t$$

Where γ usually has a value around 0,9. A visual representation of SGD without and with momentum is given in figure 11

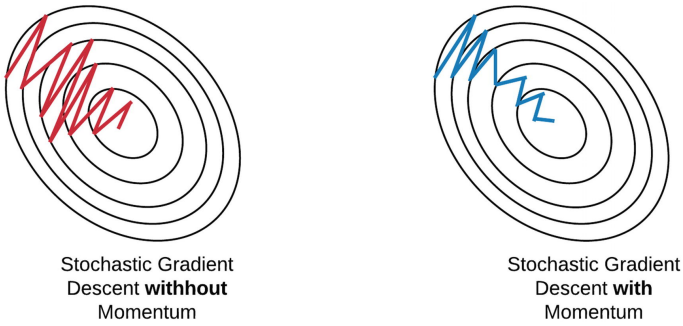

Figure 11: SGD without and with momentum. From Simone (2020).

2.3.1 Nesterov

The downside of momentum is that the algorithm tends to overshoot the minimum value of the function as it comes closer, similar to an object with a high momentum which is difficult to stop. The Nesterov optimisation algorithm solves this problem. Since the value of θ_{t+1} is determined by momentum, which is partly the direction of the gradient at time $t-1$, the position of θ_{t+1} can already be approximated. Nesterov therefore does not evaluate the gradient at time t for θ_t, but for $\theta_t - \gamma v_{t-1}$, meaning it is looking ahead when determining the gradient, enabling the algorithm to detect a possible minimum earlier.

When trying to avoid overshooting the optimum value of the objection function, the most important hyper-parameter to adjust is the learning rate ϵ. If the learning rate is too big, the algorithm will overshoot the minimum value which in the worst case can prevent the algorithm from converging. However, making the learning rate too small will cause the algorithm to take a long time before convergence is achieved. A solution is found using a dynamic learning rate, which is applied in multiple optimisation algorithms.

2.3.2 Adagrad

Adagrad is a gradient-based optimisation algorithm with a dynamic learning rate i.e. the learning rate differs per step taken by the algorithm. The goal was to start with a rather big learning rate to approach the minimum value of the objective function quicker and then decrease the learning rate as the minimum is reached to prevent overshooting. While designing the algorithm for NNs with a high dimension space for θ it turned out that, even with a dynamic learning rate, one learning rate for all parameters often let to undesirable, sub-optimal solutions. Parameters that already reached their optimal value were adjusted because other parameters were still converging. To solve the problem, Adagrad uses dynamic learning rates which differ for different parameters. Let $g_{t,i}$ denote the partial derivative of the objective function w.r.t. parameter θ_i at time step t, such that the update of θ_i with fixed learning is defined as:

$$\theta_{t+1,i} = \theta_{t,i} - \epsilon \cdot g_{t,i},$$

and the update with dynamic learning rate based on the past gradient:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\epsilon}{\sqrt{G_{t,ii} + \eta}} \cdot g_{t,i}$$

where $G_t \in \mathbb{R}^{n \times n}$ is a diagonal matrix with on index i, i the sum of the squares of the gradients w.r.t. θ_i up to time t and η a small term to avoid division by 0. The vectorized version is denoted as:

$$\theta_{t+1} = \theta_t - \frac{\epsilon}{\sqrt{G_t + \eta}} \cdot g_t$$

In practice, Adagrads dynamic learning rate solved the problem of overshooting the optimum value is most cases, but created a new problem instead. The elements on the diagonal of G_t are a sum of only positive terms, meaning that the denominator increased at every step such that the learning rate became infinitesimally small at some point. An extension on Adagrad solved this problem.

2.3.3 Adadelta

Adadelta solves the problem mentioned above by taking the average of the squared gradients over the last w time steps instead of summing the squared gradients over all past time steps. Define $E[g^2]_{t-1}$ as the average of the last w squared gradients at time $t-1$, then $E[g^2]_t$ is defined as:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2$$

Where γ has a value around 0,9. The Adadelta update rule then becomes:

$$\theta_{t+1} = \theta_t - \frac{\epsilon}{\sqrt{E[g^2]_t + \eta}} \cdot g_t$$

2.3.4 Adam

Finally, the optimisation algorithm Adam managed to combine the ideas of momentum and dynamic learning rate. Where Adadelta holds an average of the squared gradients over the last w time steps, Adam also keeps the average of the non-squared gradients over the last w time steps, similar to momentum. By doing so, Adam estimates the first moment m_t and the second moment v_t of the gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Where β_1 and β_2 have a default value of 0,9, such that the update rule is denoted as:

$$\theta_{t+1} = \theta_t - \frac{\epsilon}{\sqrt{v_t} + \eta} \cdot m_t$$

3 Research Design

3.1 Experimental Setup

The main goal of the research is to find the optimal settings of a LSTM RNN for the prediction of long-term aircraft trajectories. First of all, the architecture of the network is explored, which refers to the number of layers in the network and how they are structured. Moreover, the architecture includes the number of neurons that are found in each layer. During the research, the number of layers used as well as the neurons per layer will be experimented with to find the optimal configuration for the architecture of the network. To be precise, a LSTM RNN with one hidden layer, 2 stacked hidden layers, a bidirectional design and with an encoder-decoder model are explored. For all the configurations, the network is trained with 64, 128 and 256 neurons per layer. After the optimal architecture is found, different optimisation algorithms will be applied to find the best suited algorithm for this specific problem. The Nesterov, Adadelta and Adam optimisation algorithms will be used such that an algorithm with momentum is tested, with a dynamic learning rate and an algorithm with both. Because the batch size can be of major importance for the performance, every optimisation algorithm is tested with a batch size of 32, 64 and 128. While trying the different settings for the LSTM RNN above, the performances are measured using 3 metrices, The mean squared error (MSE), the root mean squared error (RMSE) and the mean absolute error (MAE). Defined as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}.$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|Y_i - \hat{Y}_i|.$$

As the best performing settings are selected, a more detailed analysis is performed. The errors of the predictions and filled flight plans will be plotted against the distance to the destination airport (Schiphol Amsterdam Airport), the errors of the predictions and filled flight plans will be divided in categories such as errors smaller than 5 NM, errors between 5 and 15 NM, errors between 15 and 30 NM, and errors bigger than 30 NM. the distribution of the errors will be plotted and boxplots of the errors will be shown.

3.2 The Data

The data originates from the R&D data archive of EUROCONTROL (2022a), an important authority in European aviation. The R&D data archive includes historical flight data of 4 complete months of every years; March, June, September and December. In this paper the data of 2019 of all the four available months is used. 2019 denotes the last year in European aviation which was not affected by the Covid-19 pandemic, hence the reason for choosing this year. For each of the four months, all flights departing from or arriving at European airports are included in the data. European airports refer to all the airports in the European Civil Aviation Conference (ECAC) area.

The relevant data used for this research contains all flights arriving at Schiphol International Airport, departing from a European airport. To be more precise, the interest for this research included the filed flight plans as well as the actual flight plans of the relevant flights mentioned above. The filed flight plan contains a set of coordinates, called way points, which describe the route that the aircraft should follow, the actual flight plan contains the set of way points that are actually flown. A way points is a 2-D coordinate existing of a latitude degree and a longitude degree. The data set is filtered on outliers after which 7 flights were deleted that did not depart from a European airport. Below in table 1, the most import properties of the data set are presented. Figure 12 shows the most frequently occurring airports in the data set.

Table 1: Description of the data.

category	value
Sample size (in number of flights)	67232
Range latitude (in degrees)	[23, 79]
Range longitude (in degrees)	[-28, 39]
Average length flight plan (actual/filed) (in number of way points)	26,26 / 27,19
Longest length flight plan (actual/filed) (in number of way points)	71 / 76
Shortest length flight plan (actual/filed) (in number of way points)	4 / 4
number of departure airports	311

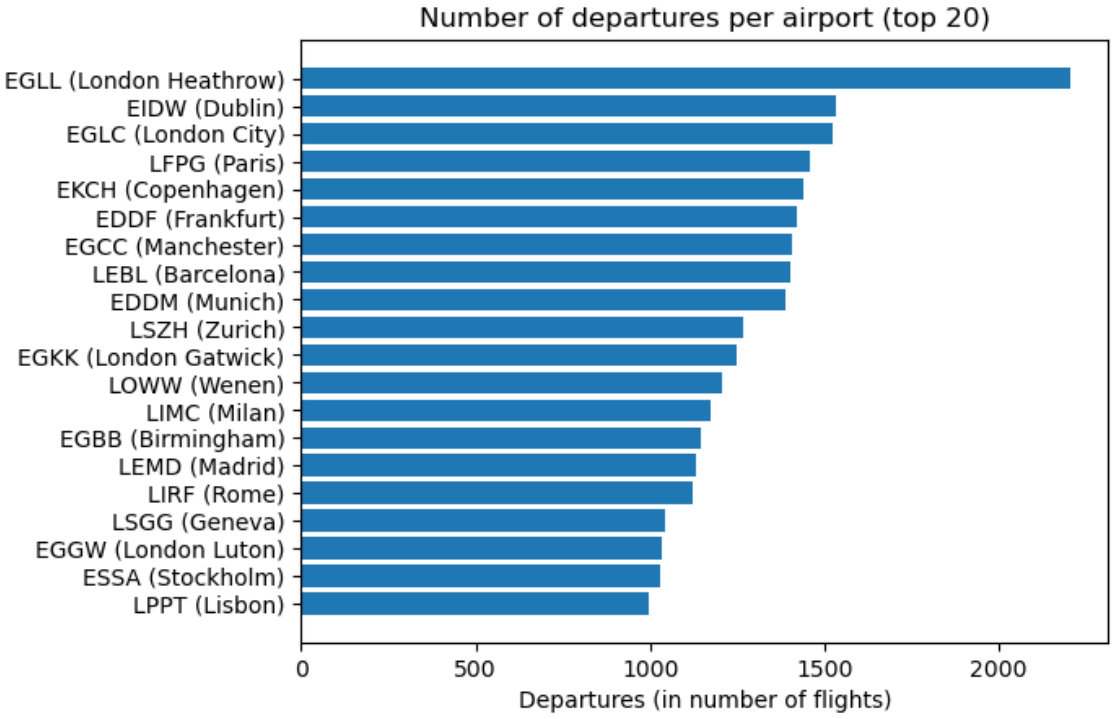

Figure 12: Most frequently occurring airports in the data set.

3.3 Network setup

The network will take the filed flight plan as input and transform it making it as similar as possible to the actual flight plan i.e. predict the deviations from the filed flight plan. Since the flight plans varied in length, padding had to be applied i.e. zeros were added to the shorter flight plans as way points such that all flight plans had equal length. A masking layer was then used to mask the added zeros such that the network cannot achieve unjustified high training accuracy for predicting the zero way points correctly. The data set in splitted in a training and test set according to a 80/20 ratio, where a seed is used to produce identical splits when testing the different parameters. Overfitting is a familiar problem in machine learning where the training data is viewed so often that the algorithm learns relations in the training data that actually do not exist, deteriorating the performance on the test data. To prevent overfitting of the network, every layer used in the network is equipped with a dropout of 0,2, meaning that every node in the layer has a 20% probability to be excluded for a given training sample that passes the layer. As with many sequence to sequence problems, this task is defined as a classification problem. This means that every time the network predicts a way point, it chooses one way point out of all the occurring way points in the data set (on 2 decimals accurate). The output layer in the network has the softmax function as activation function. Every time the network predicts a way point, the softmax function provides a probability to every occurring way point is the data set such that the network chooses the way point with the highest probability. To improve accuracy, the data is hot encoded. Hot encoding is a technique in machine learning of presenting the data set for classification problems. Every way point in the data set is presented as a vector with a dimension that equals the number of unique way points, say n. The vector then contains $n-1$ zeros and the number 1 on the remaining index. The index that contains the number 1 determines which specific way point the vector represents, meaning that there are n unique vectors for n unique way points. As standard with a network with hot encoded input, categorical crossentropy is used as cost function. For one training sample, the function is defined as:

$$C(y_i, \hat{y}_i) = -\sum_{i=1}^{k} y_i \cdot log(\hat{y}_i)$$

where y_i represents the hot encoded vector of the true value,$\hat{y}_i$ represents the hot encoded vector of the prediction and k the number of way points in the training sample.

References

Ayhan, S. and H. Samet (2016). "Aircraft Trajectory Prediction Made Easy with Predictive Analytics." In.

Berg, H. (2019). *The sigmoid function (a.k.a. the logistic function) and its derivative*. Available at https://hvidberrrg.github.io/deep_learning/activation_functions/sigmoid_function_and_derivative.html.

Bongiorno, C., G. Gurtner, F. Lillo, R. Mantegna, and R. Micciche (2016). "Statistical characterization of deviations from planned flight trajectories in air traffic management." In: *Journal of Air Transport Management* 58, pp. 152–163.

Chablani, M. (2017). *Sequence to sequence model: Introduction and concepts*. Available at https://towardsdatascience.com/sequence-to-sequence-model-introduction-and-concepts-44d9b41cd42d.

Donahue, J., L. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell (2014). "Long-term recurrent convolutional networks for visual recognition and description." In: *Journal of Air Transport Management* 58.

Eapen, J., D. Bein, and A. Verma (2019). "Novel Deep Learning Model with CNN and Bi-Directional LSTM for Improved Stock Market Index Prediction." In: *IEEE 9th Annual Computing and Communication Workshop and Conference*, pp. 264–270.

Eliasy, A. and J. Przychodzen (2020). "The role of AI in capital structure to enhance corporate funding strategies". In: *Array* 6, p. 100017.

EUROCONTROL (2022a). *COVID-19 impact on the European air traffic network.* Available at https://www.eurocontrol.int/covid19.

— (2022b). *RD data archive*. Available at https://www.eurocontrol.int/dashboard/rnd-data-archive.

floyd (2019). *Can we use the folded version of RNN in training?* Available at https://stats.stackexchange.com/q/419269.

Graves, A. and N. Jaitly (2014). "Towards end-to-end speech recognition with recurrent neural networks." In: *Proceedings of the 31st International Conference on Machine Learning*, pp. 1764–1722.

Hochreiter, S. and J. Schmidhuber (1997). "Long Short-Term Memory." In: *Neural Computation* 9, pp. 1735–1780.

J., Ryan T. J. (2020). *LSTMs Explained: A Complete, Technically Accurate, Conceptual Guide with Keras*. Available at https://medium.com/analytics-vidhya/lstms-explained-a-complete-technically-accurate-conceptual-guide-with-keras-2a650327e8f2.

Jia, P., H. Chen, L. Zhang, and D. Han (2022). "Attention-LSTM Based Prediction Model for Aircraft 4-D Trajectory." In: *Research Square*.

Klosowski, P (2018). "Deep Learning for Natural Language Processing and Language Modelling." In: *Journal of Air Transport Management* 58, pp. 223–228.

Liu, Y. and M. Hansen (2018). "Predicting Aircraft Trajectories: A Deep Generative Convolutional Recurrent Neural Networks Approach." In: *Institutes of Transportation Studies*.

Lv, J., Q. Li, and X. Wang (2017). "Statistical characterization of deviations from planned flight trajectories in air traffic management." In: *Big Data and Smart Computing*, pp. 82–89.

Ma, L. and S. Tian (2020). "A Hybrid CNN-LSTM Model for Aircraft 4D Trajectory Prediction." In: *IEEE Access*, pp. 134668–13680.

Melcher, K. (2021). *A Friendly Introduction to [Deep] Neural Networks*. Available at https://www.knime.com/blog/a-friendly-introduction-to-deep-neural-networks.

Murça, M., R. Delaura, R. Hansman, R. Jordan, T. Reynolds, and H. Balakrishnan (2016). "Trajectory Clustering and Classification for Characterization of Air Traffic Flows". In.

Naessens, H., T. Philip, M. Piatek, K. Schippers, and R. Parys (2017). *Predicting flight routes with a Deep Neural Network in the operational Air Traffic Flow and Capacity Management system.* Tech. rep. No. 37. Maastricht, The Netherlands: EUROCONTROL Maastricht Upper Area Control Centre.

Pang, Y., H. Yao, J. Hu, and Y. Liu (2019). "A Recurrent Neural Network Approach for Aircraft Trajectory Prediction with Weather Features From Sherlock". In.

Rodriguez, E. (2022). "Effect of Flexible Use of Airspace Availability and Plannability on Fuel Efficiency". PhD thesis. Delft University of Technology.

Ruder, S. (2016). *An overview of gradient descent optimization algorithms*. Available at https://ruder.io/optimizing-gradient-descent/.

Salaun, E., M. Gariel, A. Vela, and E. Feron (2012). "Review Of Trajectory Accuracy Methodology And Comparison Of Error Measurement Metrics." In: *Journal of Guidance, Control and Dynamics* 35(2), pp. 563–577.

Shi, Z., M. Xu, Q. Pan, B Yan, and H. Zhang (2007). "LSTM-based Flight Trajectory Prediction." In: *International Joint Conference on Neural Networks*.

Simone (2020). *Stochastic Gradient Descent on your microcontroller*. Available at https://eloquentarduino.github.io/2020/04/stochastic-gradient-descent-on-your-microcontroller/.

Zhang, X. and S. Mahadevan (2020). "Bayesian neural networks for flight trajectory prediction and safety assessment." In: *Decision Support Systems* 131, pp. 113–246. ISSN: 0167-9236.

Zhengfeng, X., Z. Weili, C. Xiao, and C. Puwen (2021). "Multi-Aircraft Trajectory Collaborative Prediction Based on Social Long Short-Term Memory Network". In: *Aerospace* 8, p. 115.

Zvornicanin, E. (2022). *Differences Between Bidirectional and Unidirectional LSTM*. Available at https://www.baeldung.com/cs/bidirectional-vs-unidirectional-lstm.