Forecasting the USD/SEK exchange rate using deep neural networks

Thomas Hamfelt

Bachelor's thesis
2019:K22

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

CENTRUM SCIENTIARUM MATHEMATICARUM

Forecasting the USD/SEK exchange rate using deep neural networks

Thomas Hamfelt
BSc Thesis in Mathematical Statistics
Lund University

August 15, 2019

Abstract

This thesis is about predicting the average ten minute closing bid price of the USD/SEK exchange rate by applying deep learning methods. First, the time lag method is applied for the vanilla Feedforward Neural Network (FNN) to undertake one-step prediction. Secondly, three univariate Long Short-Term Memory (LSTM) models are used to undertake one-step and multi-step prediction. Each network is theoretically described and motivated.

The results indicate that both the FNN and LSTM are applicable to time series prediction and that the LSTM outperforms the FNN. Furthermore, the results suggests the LSTM can outperform the naïve predictor by a small margin but it remains uncertain. It is concluded that to detect structure in the exchange rate much more computing power might be required to learn from significantly longer time series as input.

Finally, some economic theory is reviewed and presented which could be used as potential inputs improving the results. A small discussion on overlooked biases is also included.

Acknowledgements

I would like to thank my supervisors Alexandros Sopasakis and Henning Petzka for their support and guidance. This thesis has been my introduction to deep learning and their comments, feedback and suggestions have been of great value.

Contents

Chapter 1

Introduction

In the daily news we can read about the development of currencies. And this is for a good reason. The market for currency, also known as the *foreign exchange (FX)* market, is the worlds most actively traded market. On an average day approximately $5 trillion changes hands, which is 25 times more than the market for equities [5]. The market is of great importance to the global economy and it affects states, corporations as well as individuals. Naturally, it is of interest to estimate the future exchange rates. This thesis investigates whether the Feedforward Neural Network (FNN) and the Long-Short Term Memory (LSTM) network can be trained and tested on USD/SEK data consisting of ten minute averages. The data is from May 2016 until May 2019.

The second chapter introduces the theory behind the FNN. Following, the LSTM will be introduced and motivated. As the FNN is the most essential algorithm for deep learning, its inner workings are covered in great detail. Subsequently, the introduction of recurrent neural networks is motivated and the LSTM is presented. In the third chapter, the data is presented and visualized. In chapter four and five, the FNN and the LSTM are implemented and used to predict the USD/SEK exchange rate. One FNN model is used for one-step prediction. Three LSTM models are included where two undertake one-step predictions but with varying lengths of input data. The third and last perform multi-step prediction. In the sixth and final chapter results are discussed and conclusions put forward. Some economic theory is reviewed which could be the motivation for additional inputs which might improve the results. The suggestions are mainly concerned with the interest rate parity condition.

1.1 Software

This thesis used Python as programming language for coding and leveraged the packages Scikit-learn and Keras (Tensorflow backend) when implemeting LSTM networks. In terms of hardware, a standard household computer has been used with the exception of some implementations being done in Google Colaboratory. Google Colaboratory offers free access to a TESLA K80 GPU which has improved the speed of training networks. When Google Colaboratory has been used it will be indicated.

Chapter 2

Theory

2.1 Feedforward Neural Network (FNN)

The *Feedforward Neural Network (FNN)* is the most fundamental of deep learning models. The purpose of the FNN is to approximate some function $\mathbf{y} = f(\mathbf{x}; \mathbf{\Theta})$ by finding optimal parameters $\mathbf{\Theta}$ [11]. The parameters of a FNN are referred to as *weights*. FNNs are called networks as they consist of several layers of *nodes*. The first layer is known as the *input layer* and the final as the *output layer*. Any layer, except the input and output layers, is called a *hidden layer*. Each node in the first layer represents one dimension of the input to the network. The inputs in the first layer can be represented as a vector, $\mathbf{x} \in \mathbb{R}^n$. Subsequently, $\mathbf{x}$ is is mapped linearly to the second layer by multiplication with a matrix, $\mathbf{W}^1_{m\times n}$, which is known as the first weight matrix. In order to enable the network to not only model linear transformations of the input, each node in the second layer is evaluated using a non-linear differentiable function. This function is referred to as an *activation function*. In Section 2.4 activation functions will be covered in more detail. The nodes in the second layer, after the non-linear evaluation, can be represented by a vector $\mathbf{v} \in \mathbb{R}^m$. Depending on the number of hidden layers, this process is repeated for each hidden layer. If we restrict our case to only one hidden layer and one node in the output layer, then $\mathbf{v}$ is mapped to the output layer through the second weight matrix $\mathbf{W}^2_{1\times m}$ and flushed through an activation function. This set up is illustrated in Figure 2.1 where the input has four dimensions, the hidden layer ten and the output one. The output must not be one dimensional however but could also be multivariate.

2.2 Feedforward- and Backpropagation

The process described in the previous section is referred to as *feedforward propagation*. It is the process where the network receives some input, $\mathbf{x}$ and produces some output $\hat{y}$. Once the network has produced the output the true value, y, is revealed and some error metric is computed. Next, this error information is fed backwards through the network. This process is called *backpropagation*. Backpropagation (also referred to as backprop) is the main workhorse of the FNN. By taking the partial derivative of the output error metric with respect to each weight in the network, the gradient of each layer's weight matrix can be computed. Thereafter, by multiplying the gradient with a small scalar η, $\eta > 0$ we can subtract the product from the weight matrix. This makes the error produced at the output smaller. By perform-

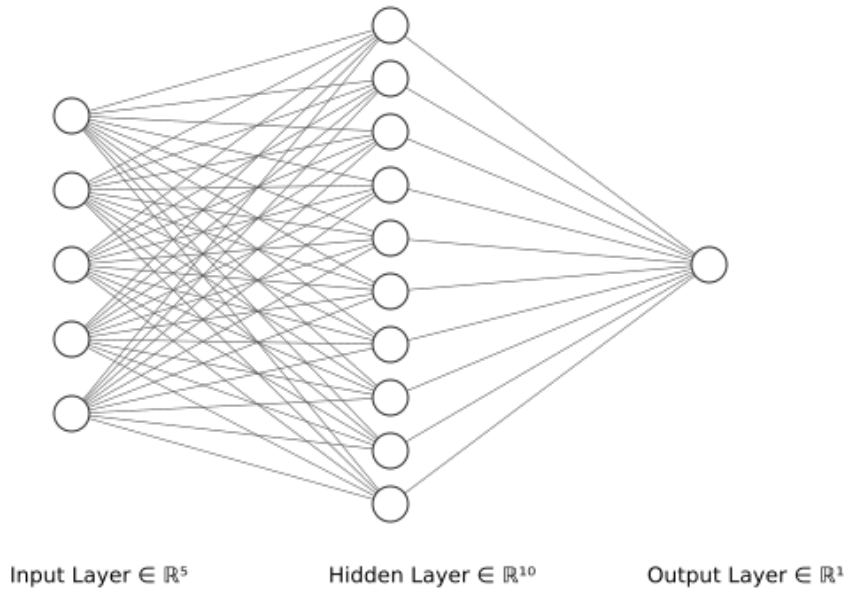

Figure 2.1: Illustration of a Feedforward Neural Network (FNN). Note the dimensionality of the input layer, hidden layer and the output layer.

ing this update iteratively, we eventually reach a (local or global) minimum of the loss as a function of the weight parameters. This process is referred to as *gradient descent*. In other words, it is the process which enables the network to learn from its mistakes and make adjustments for future input. Next, the backpropagation algorithm will be derived in full detail after some more rigid definition of the network architecture. The set up and presentation of the derivation was originally done by Du and Swamy [8].

Let there be $1, \ldots, M$ layers in the network. Each layer contains J_m, $m = 1, \ldots, M$ nodes. Denote the matrix of weights from layer $(m-1)$ to m by $\mathbf{W}^{(m-1)}$, $m = 2, \ldots, M$. For the p'th data example, the bias and output of the i'th neuron in the m'th layer are referred to as $\theta_i^{(m)}$ and $o_{p,i}^{(m)}$. Expressed in vector format for a layer,

$$\boldsymbol{\theta}^{(m)} = \left[\theta_1^{(m)}, \ldots, \theta_{J_m}^{(m)}\right]^T,$$

$$\mathbf{o}_p^{(m)} = \left[o_{p,1}^{(m)}, \ldots, o_{p,J_m}^{(m)}\right]^T.$$

Furthermore also for the p'th example and for layers $m = 2, \ldots, M$

$$\mathbf{o}_p^{(m)} = \phi^{(m)}\left(\mathbf{net}_p^{(m)}\right),$$

$$\mathbf{net}_p^{(m)} = \left[\mathbf{W}^{(m-1)}\right]^T \mathbf{o}_p^{(m-1)} + \boldsymbol{\theta}^{(m)},$$

$$\mathbf{net}_p^{(m)} = \left[net_{p,1}^{(m)}, \ldots, net_{p,J_m}^{(m)}\right]^T.$$

where $\phi^{(m)}$ is the activation function which is applied element wise.
The input and output of the network for the p'th data example are

$$\mathbf{o}_p^{(1)} = \mathbf{x}_p, \quad \hat{\mathbf{y}}_p = \mathbf{o}_p^{(M)}.$$

Let $(\mathbf{x}_p, \mathbf{y}_p) \in S$ be every a pair of the training data where $p = 1, \ldots, N$ where N is the number of training pair. Denote the objective cost function, which is to be optimized, by E and let it be the mean squared error such that

$$E = \frac{1}{N}\sum_{p\in S} E_p = \frac{1}{2N}\sum_{p\in S} ||\hat{\mathbf{y}}_p - \mathbf{y}_p||^2 \tag{2.1}$$

where $\hat{\mathbf{y}}_p$ is the output of the FNN. Multiplying with $\frac{1}{2}$ is a mathematical construct which will later ease the notation after differentiation. Furthermore denote,

$$E_p = \frac{1}{2}||\hat{\mathbf{y}}_p - \mathbf{y}_p||^2 = \frac{1}{2}\mathbf{e}_p^T\mathbf{e}_p \tag{2.2}$$

and

$$e_p = \hat{y}_p - y_p. \tag{2.3}$$

The goal now is to minimize E through gradient descent. In order to do this, the gradient must be computed and updated accordingly to

$$\Delta_p \mathbf{W} = -\eta \frac{\partial E_p}{\partial \mathbf{W}}.$$

Let the weights from layer $m-1$ to m, $m = 2, \ldots, M$ be elements in the matrix $\mathbf{W}^{(m)} = [w_{ij}^{(m)}]$. Proceed by applying the chain rule,

$$\frac{\partial E_p}{\partial w_{uv}^{(m)}} = \frac{\partial E_p}{\partial net_{p,v}^{(m+1)}} \frac{\partial net_{p,v}^{(m+1)}}{\partial w_{uv}^{(m)}}. \tag{2.4}$$

Note again that the subscript p, v indicate the v'th node in the p'th training example in $p = 1, \ldots, N$. The second factor in (2.4) can be rewritten to

$$\frac{\partial net_{p,v}^{(m+1)}}{\partial w_{uv}^{(m)}} = \frac{\partial}{\partial w_{uv}^{(m)}}\left[\sum_{k=1}^{J_m} w_{kv}^{(m)} o_{p,k}^{(m)} + \theta_v^{(m+1)}\right] = o_{p,u}^{(m)}. \tag{2.5}$$

Next, the chain rule can be applied to the first factor in (2.4)

$$\frac{\partial E_p}{\partial net_{p,v}^{(m+1)}} = \frac{\partial E_p}{\partial o_{p,v}^{(m+1)}} \frac{\partial o_{p,v}^{(m+1)}}{\partial net_{p,v}^{(m+1)}}. \tag{2.6}$$

Recalling that $o_{p,v}^{(m+1)} = \phi^{(m+1)}(net_{p,v}^{(m+1)})$, (2.6) can be rewritten to

$$\frac{\partial E_p}{\partial net_{p,v}^{(m+1)}} = \frac{\partial E_p}{\partial o_{p,v}^{(m+1)}} \phi_v'^{(m+1)}\left(net_{p,v}^{(m+1)}\right). \tag{2.7}$$

Subsequently, two cases will be made for the first factor in (2.7). In the first case $m = M - 1$ (last layer) and in the second case $m = 1, \ldots, M - 2$ (non-last layers).

By letting $m = M - 1$ and substituting E_p we obtain

$$\frac{\partial E_p}{\partial o_{p,v}^{(m+1)}} = \frac{\partial}{\partial o_{p,v}^{(m+1)}} \left[\frac{1}{2} \sum_{k=1}^{J_{m+1}} (o_{p,k}^{(m+1)} - y_k)^2 \right] = (o_{p,v}^{(m+1)} - y_v) = e_{p,v}. \tag{2.8}$$

And by letting $m = 1, \ldots, M - 2$

$$\begin{aligned} \frac{\partial E_p}{\partial o_{p,v}^{(m+1)}} &= \sum_{k=1}^{J_{(m+2)}} \left[\frac{\partial E_p}{\partial net_{p,k}^{(m+2)}} \frac{\partial net_{p,k}^{(m+2)}}{\partial o_{p,v}^{(m+1)}} \right] \\ &= \sum_{k=1}^{J_{(m+2)}} \left[\frac{\partial E_p}{\partial net_{p,k}^{(m+2)}} \frac{\partial}{\partial o_{p,v}^{(m+1)}} \left[\sum_{i=1}^{J_{m+1}} w_{ik}^{(m+1)} o_{p,j}^{(m+1)} + \theta_k^{(m+2)} \right] \right] \\ &= \sum_{k=1}^{J_{(m+2)}} \left[\frac{\partial E_p}{\partial net_{p,k}^{(m+2)}} w_{vk}^{(m+1)} \right]. \end{aligned} \tag{2.9}$$

Next, define the function $\delta_{p,v}^{(m)}$ from (2.7)

$$\delta_{p,v}^{(m)} = -\frac{\partial E_P}{\partial net_{p,v}^{(m)}}, \quad m = 2, \ldots, M. \tag{2.10}$$

By plugging in (2.8)

$$\delta_{p,v}^{(M)} = -e_{p,v} \phi_v'^{(M)} \big(net_{p,v}^{(M)} \big), \quad m = M - 1. \tag{2.11}$$

And by plugging in (2.9)

$$\delta_{p,v}^{(m+1)} = \phi_v'^{(m+1)} \big(net_{p,v}^{(m+1)} \big) \sum_{k=1}^{J_{(m+2)}} \left[\delta_{p,k}^{(m+2)} w_{vk}^{(m+1)} \right], \quad m = 1, \ldots, M - 2. \tag{2.12}$$

Thus, it has been shown that $\delta_{p,v}^{(m+1)}$ can be solved throughout the whole network by first applying (2.11) and then recursively by applying (2.12). Finally, by using the results from (2.10) and (2.5), (2.4) can be rewritten as

$$\frac{\partial E_p}{\partial w_{uv}^{(m)}} = -\delta_{p,v}^{(m+1)} o_{p,u}^{(m)} \tag{2.13}$$

which enable us to update the weights and undertake gradient descent.

One can also update the bias terms using the above results. For $m = 2, \ldots, M$

$$\begin{aligned} \frac{\partial E_p}{\partial \theta_j^{(m)}} &= \frac{\partial E_p}{\partial o_{p,j}^{(m)}} \frac{\partial o_{p,j}^{(m)}}{\partial net_{p,j}^{(m)}} \frac{\partial net_{p,j}^{(m)}}{\partial \theta_j^{(m)}} \\ &= \frac{\partial E_p}{\partial o_{p,j}^{(m)}} \frac{\partial o_{p,j}^{(m)}}{\partial net_{p,j}^{(m)}} \frac{\partial}{\partial \theta_j^{(m)}} \left[\sum_{k=1}^{J_{m-1}} w_{kj}^{(m-1)} o_{p,k}^{(m-1)} + \theta_j^{(m)} \right] \\ &= \frac{\partial E_p}{\partial o_{p,j}^{(m)}} \frac{\partial o_{p,j}^{(m)}}{\partial net_{p,j}^{(m)}} \end{aligned} \tag{2.14}$$

which is the same as in (2.7). Thus we can use (2.8) and (2.9) to update the biases.

2.3 Universal Approximation Theorem

The goal of a neural network is to approximate a given function. It has been shown neural networks can approximate a wide range of interesting functions. In 1989, George Cybenko provided a theorem showing that a two layered FNN, using a sigmoid activation function, can approximate continuous functions on compact subsets of higher dimensions. The *Universal Approximation Theorem* provided by Cybenko is stated below [6].

Theorem 1 (Universal Approximation Theorem) *Let I_n denote the n-dimensional unit cube, $[0,1]^n$. The space of continuous functions on I_n is denoted by $C(I_n)$. Let σ be any continous sigmoidal function. The finite sums of the form*

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j^T x + b_j)$$

are dense in $C(I_n)$. In other words, given any

$$f \in C(I_n) \text{ and } \epsilon > 0$$

there is a sum, $G(x)$, of the above form, for which

$$|G(x) - f(x)| < \epsilon \text{ for all } x \in I_n.$$

The theorem does not touch upon matters such as when the network will converge or the actual size of the network necessary, only that its possible. There are extensions to the theorem. Sonoda and Murata were able to prove that the theorem also holds when using unbounded activation functions, such as the ReLU (see Section 2.4) [14].

2.4 Activation Functions

In order to undertake gradient descent, it has traditionally been argued that a strictly monotonically increasing differentiable function is needed [8]. Furthermore, the introduction of activation functions breaks the linearity of the system and enables the neaural network to make a non-linear approximation [11]. Well established activation functions are the hyperbolic tangent (2.15) and the sigmoid (2.16).

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.15}$$

$$S(x) = \frac{e^x}{e^x + 1} \tag{2.16}$$

It has however been suggested that the notion of differentiability can be relaxed and arguments are being voiced that the rectified linear unit (ReLU) function (2.17) is the preferred choice [11].

$$ReLU(x) = max(0, x) \tag{2.17}$$

The fact that the derivative of the ReLU function is not defined at zero does not stop it from being used. When implemeneted, the derivative at $x = 0$ is usually

set to zero. On the second row in Figure 2.2 the derivatives of the three activation functions have been plotted for $x = [-10, 10]$. The tanh and sigmoid derivatives go to zero as $|x| \to \infty$. This can hinder the learning capability of the neural network as there is less information in the gradient. A phenomenom known as *saturation*. The ReLU does not suffer from this. It benefits from its constant derivative of one when $x > 0$. Another benefit comes from simpler computational complexity. The ReLU is not flawless however as neurons can get stuck on the negative side and only output zero. This problem is known as *dying ReLU* [13] .

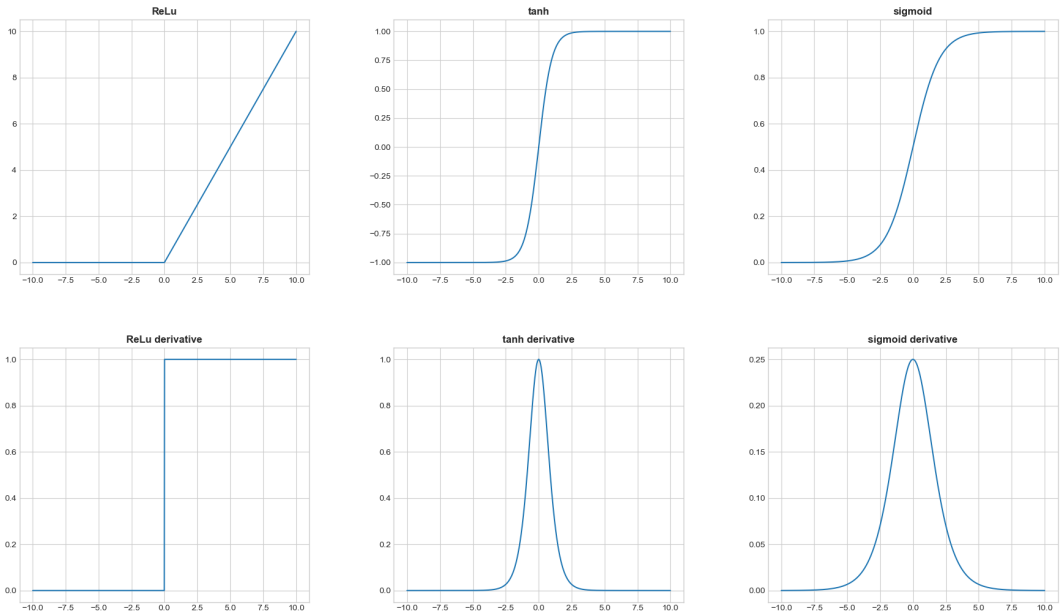

Figure 2.2: Three common activation functions and their derivatives. Note that the ReLU function does not have a defined derivative at the origin. But this does not stop it from being implemented. One advantage of the ReLU function compared to the tanh and sigmoid is that is does not suffer from *saturation*. Saturation comes from as $|x| \to \infty$, the tanh and sigmoid derivatives go to zero. The ReLU function is not flawless however. When applied, the neurons can get stuck with $x < 0$ and only output zero. A problem known as *dying ReLU*.

2.5 Data Preprocessing

Before submitting data to a neural network, it is often best to preprocess it. By centering the values around zero, numerical conditions are improved and the gradient descent can be undertaken more easily [3]. Meanwhile it is important to keep in mind the range of the output layer's activation function. Overall, there exists no consensus on whether if and how to preprocess and often it is determined case-by-case. This thesis will be apply two types of preprocessing techniques which are covered below.

2.5.1 [0,1] Transformation

One common linear transformation which is known as *normalization* is to map the data to a [0,1] interval. Let x_{min} and x_{max} denote the smallest and biggest observations in all the data (training and testing). Let x_i denote the i'th observtion in all the data (training and testing). The data can then be normalized through

$$x_{norm} = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

where x_{norm} is the normalized value.

It is a suitable transformation when either the sigmoid or ReLU function is used as activation function in the output layer. On the downside, the transformation is sensitive to outliers [12].

2.5.2 log & Zero Mean Transformation

In traditional time series modelling, the log & zero mean transformation is a common transform to apply before undertaking any analysis. First, the log transformation which is a non-linear transformation stabilizes the variance when it is considered not to be stationary. When applied in deep learning, it makes the network measure errors in terms of ratios of the original values instead of differences since

$$log(\frac{a}{b}) = log(a) - log(b).$$

This can be a desired option when the data has a high variance. For example, if the true value of the data is 10 and the network predicts 5, the squared difference is 25. The same difference is obtained if the true value is 1000 and the network predicts 995. Yet the later relative prediction error is much smaller than the earlier one and we would not want the network to treat the error as the same. This issue is mitigated by log-transformation as $log(995/1000)^2 << log(5/10)^2$ [2].

Moreover, the log-transformation is bijective and keeps the ordering. This makes it possible to rescale the data after analysis to original values.

Finally, by subtracting the mean and making it zero mean, numerical conditions are improved.

2.6 Loss Function

Loss functions, or objective cost functions, are used by the network to evaluate how well it performs. This thesis applies two types of loss metrics.

2.6.1 Mean Squared Error (MSE) and Root Mean Squared Error (RMSE)

One of the most popular loss functions within the field is the MSE,

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

By squaring the error, prediction errors which are larger are penalized more than smaller ones. This is of interest when the proportionality of the error is not linear. One can also take the square root of the MSE and obtain the RMSE.

2.6.2 Mean Absolute Error (MAE)

Another popular loss function is the MAE,

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

The MAE does not care about the direction of the error since it is the sum of absolute error values. Opposite to the MSE and RMSE, the MAE treats the error proportionality as linear. It could be argued that this loss is most appropriate to this thesis. It is hard to motivate why prediction errors of an exchange range should not be treated linearly.

2.7 Recurrent Neural Networks (RNNs)

When using the FNN for any type of temporal sequences, such as time series, one major drawback is that it processes each sample independently. This brought about the introduction of the *Recurrent Neural Network (RNN)*. Instead of letting the hidden state at time t only depend on the input at time t, it also depends on the hidden state at time $t-1$. It can be viewed as an internal loop where information at time $t-1$ can flow to time t, enabling the neural network to have memory.

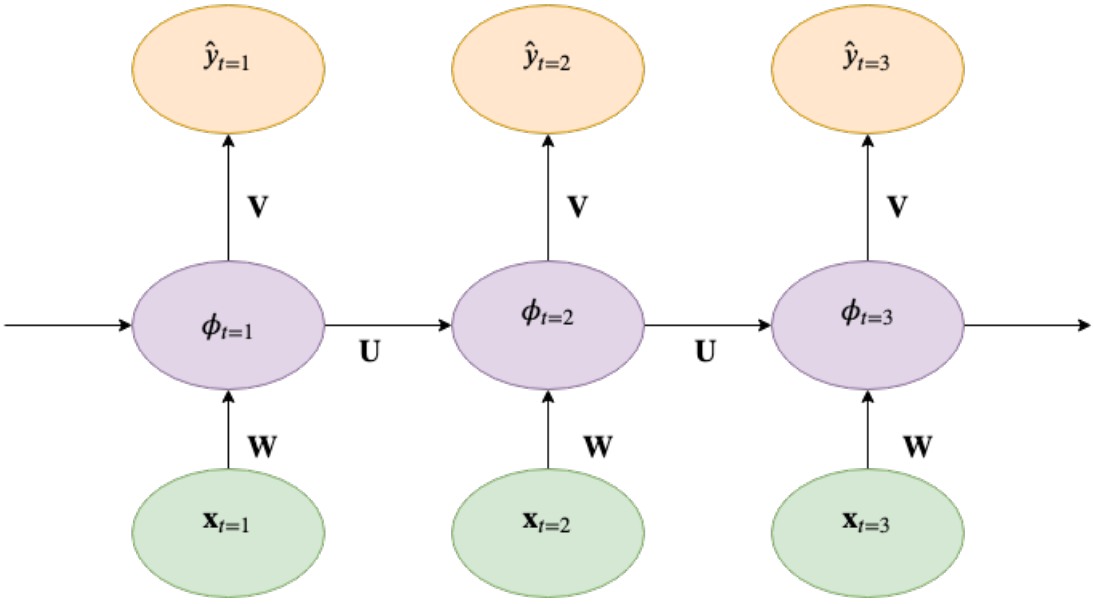

Figure 2.3: The internal loop of a RNN visualized and unrolled. Similar to the FNN, there is an input layer ($\mathbf{x}_t$) and hidden layer ($\boldsymbol{\phi}_t$). Note the new connection between hidden layers ($\mathbf{U}_t$). This enables the RNN to have memory.

In Figure (2.3), let $\mathbf{x}_t \in \mathbb{R}^n$ in (2.3) denote the input vector at time t. Let ϕ_t denote the activation of the hidden layer consisting on m nodes at time t. And let

$\hat{y}_t$ be the network's output at time t. As in the FNN, there exists a weight matrix connecting the input- and hidden layer, $\mathbf{W}_{nxm}$. What sets the RNN apart from the FNN is the introduction of a matrix $\mathbf{U}_{mxm}$ which connects the hidden layers as well. Thus the activation of a hidden layer at time t can be expressed as $\phi(\mathbf{W}^T\mathbf{x}_t+\mathbf{U}\boldsymbol{\phi}_{t-1})$ where ϕ denotes the activation function. Note that bias terms have been omitted for simplicity.
The RNN uses backpropogation to compute the gradients but since it's done over several time steps it is sometimes referred to as *Backpropagation Through Time (BPTT)*. A toy example, inspired from [1], will illustrate the challenges which appear when backpropagation is done over several time steps.
In Figure 2.3, let $x_t = \mathbf{x}_t$, $\hat{y}_t = \hat{\mathbf{y}}_t$, $w = \mathbf{W}, u = \mathbf{U}$ and $v = \mathbf{V}$. Thus the same setup as in Figure 2.3 but one dimensional. Let the error at time t be $E_t = \frac{1}{2}(\hat{y}_t - y_t)^2$. For illustrative purposes, we will backpropagate from $t = 3$. To update v, u and w we must differentiate the error E_3 which is a composition of several functions. First we differentiate with respect to v,

$$\begin{aligned}\frac{\partial E_3}{\partial v} &= \frac{\partial}{\partial v}(\frac{1}{2}(\hat{y}_3 - y_3)^2) \\ &= \frac{\partial}{\partial v}(\frac{1}{2}(v\phi_3 - y_3)^2) \\ &= (v\phi_3 - y_3)\phi_3 \\ &= (\hat{y}_3 - y_3)\phi_3.\end{aligned}$$

To differentiate with respect to u and w we first set up the function composition according to Figure 2.3

$$E_3(\hat{y}_3(\phi_3(u, w, \phi_2(u, w, \phi_1(u, w))))).$$

Next, by applying the total derivative chain rule we differentiate with respect to u and w and obtain

$$\begin{aligned}\frac{\partial E_3}{\partial u} &= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial \phi_3}(\frac{\partial \phi_3}{\partial u} + \frac{\partial \phi_3}{\partial \phi_2}(\frac{\partial \phi_2}{\partial u} + \frac{\partial \phi_2}{\partial \phi_1}\frac{\partial \phi_1}{\partial u})) \\ &= \frac{\partial E_3}{\partial \hat{y}_3}(\frac{\partial \hat{y}_3}{\partial \phi_3}\frac{\partial \phi_3}{\partial u} + \frac{\partial \hat{y}_3}{\partial \phi_3}\frac{\partial \phi_3}{\partial \phi_2}\frac{\partial \phi_2}{\partial u} + \frac{\partial \hat{y}_3}{\partial \phi_3}\frac{\partial \phi_3}{\partial \phi_2}\frac{\partial \phi_2}{\partial \phi_1}\frac{\partial \phi_1}{\partial u})\end{aligned}$$

$$\begin{aligned}\frac{\partial E_3}{\partial w} &= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial \phi_3}(\frac{\partial \phi_3}{\partial w} + \frac{\partial \phi_3}{\partial \phi_2}(\frac{\partial \phi_2}{\partial w} + \frac{\partial \phi_2}{\partial \phi_1}\frac{\partial \phi_1}{\partial w})) \\ &= \frac{\partial E_3}{\partial \hat{y}_3}(\frac{\partial \hat{y}_3}{\partial \phi_3}\frac{\partial \phi_3}{\partial w} + \frac{\partial \hat{y}_3}{\partial \phi_3}\frac{\partial \phi_3}{\partial \phi_2}\frac{\partial \phi_2}{\partial w} + \frac{\partial \hat{y}_3}{\partial \phi_3}\frac{\partial \phi_3}{\partial \phi_2}\frac{\partial \phi_2}{\partial \phi_1}\frac{\partial \phi_1}{\partial w})\end{aligned}$$

The example makes it apparent that if backpropagation is done over several time steps, more derivatives will be computed and appear as factors in the expressions. One documented side effect from this is the *vanishing- or exploding gradient* problem. The vanishing gradient problem comes from the fact that in common activation functions such as the tanh and sigmoid, the derivatives are between 0 and 1 (see the second row in Figure 2.2). When more small factors are multiplied together, the

product will become extremely small and the gradient information of several steps starts to vanish. This makes it difficult for the RNN to learn long sequences. On the other hand, if we assume the activation function to be linear then the weights u and w will be factors in the expression above. If they are larger than 1, then this can cause the gradient to explode. Hence this situation is referred to as the exploding gradient problem. In 1991 Sepp Hochreiter was the first one to describe these issues [1].

In 1997, a new kind of RNN was introduced by Hochreiter and Schmidhuber named the *Long Short Term Memory (LSTM)* network. It was introduced to combat the vanishing- or exploding gradient problem in the vanilla RNN. The following presentation of the LSTM was originally presented by Lipton, Berkowitz and Elkan [10]. In the LSTM, the nodes of the network (see Figure 2.1) have been exchanged with *memory cells*. One memory cell is illustrated in Figure 2.4. It is a unit in itself and is built up from simpler nodes connected in a special pattern. The operations will be covered in more detail below. Please note that vector representation, such as $\mathbf{s}$, indicates the nodes of the entire layer of memory cells. When the subscript c is used, for example s_c, it indicates the components of memory cell c. The superscript (t) indicates time. The letter Π indicates multiplication between nodes.

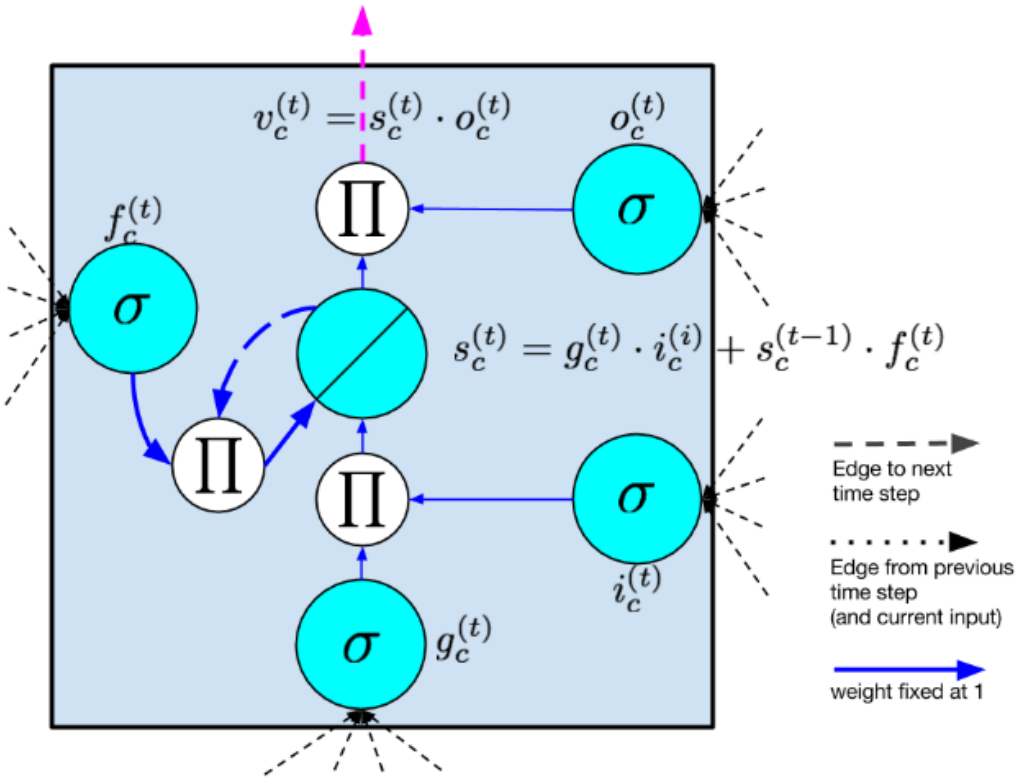

Figure 2.4: LSTM memory cell illustrated from [10]. In an LSTM, the memory cells replace the nodes. The input gate, $g_c^{(t)}$, can be seen at the bottom. The input gate, $i_c^{(t)}$, is at the bottom-right. The internal state, $s_c^{(t)}$, is in the middle. The forget gate, $f_c^{(t)}$, is located to the middle-left. The output gate is at the top-right $o_c^{(t)}$.

- **Input node**: Denoted $g_c^{(t)}$ the input node is similar to a node in the hidden

layer of the vanilla RNN. It has, the input layer $\mathbf{x}^{(t)}$ and the output from previous memory cells $\mathbf{h}^{(t-1)}$, as input. In the original paper, the activation function was a sigmoid but in today's LSTMs a tanh is used.

- **Input gate**: Denoted by $i_c^{(t)}$ the input gate also has the input layer $\mathbf{x}^{(t)}$ and the output from previous memory cells $\mathbf{h}^{(t-1)}$, as input. It uses a sigmoid function as activation function and is called a gate since its multiplied with the input node. Considering the sigmoid function in the input gate, the input node is multiplied with a value between 0 and 1. Zero implies nothing is passed through the gate while a one lets everything through.

- **Internal state**: Denoted by $s_c^{(t)}$, the internal state is located at the center of the memory cell. It is self-connected to other adjacent memory cells with constant weights.

- **Forget gate**: Denoted by $f_c^{(t)}$ and smilar to the input gate, the forget gate controls the input to cell state. It is a way for the network to reset its internal memory.

- **Output gate**: Denoted by $o_c^{(t)}$ the output gate is multiplied with the output of the cell state to produce the final output of the memory cell, $v_c^{(t)}$. It is common that the internal state first is flushed through the tanh function. This yields that the output of the memory cell is in the range of the tanh function.

For completeness, the equations for the full LSTM algorithm are listed below. Please note that the superscript of weight matrices, for example W^{gx}, indicates the memory cell's component (g) and the input (x). Let ϕ denote the tanh function and σ the sigmoid function. Biases are denoted by $\mathbf{b}$.

$$\mathbf{g}^{(t)} = \phi(W^{gx}\mathbf{x}^{(t)} + W^{gh}\mathbf{h}^{(t-1)} + \mathbf{b}_g)$$

$$\mathbf{i}^{(t)} = \sigma(W^{ix}\mathbf{x}^{(t)} + W^{ih}\mathbf{h}^{(t-1)} + \mathbf{b}_i)$$

$$\mathbf{f}^{(t)} = \sigma(W^{fx}\mathbf{x}^{(t)} + W^{fh}\mathbf{h}^{(t-1)} + \mathbf{b}_f)$$

$$\mathbf{o}^{(t)} = \sigma(W^{ox}\mathbf{x}^{(t)} + W^{oh}\mathbf{h}^{(t-1)} + \mathbf{b}_o)$$

$$\mathbf{s}^{(t)} = \mathbf{g}^{(t)} \odot \mathbf{i}^{(t)} + \mathbf{s}^{(t-1)} \odot \mathbf{f}^{(t)}$$

$$\mathbf{h}^{(t)} = \phi(\mathbf{s}^{(t)}) \odot \mathbf{o}^{(t)}$$

The output of the entire LSTM layer at time t is denoted by $\mathbf{h}^{(t)}$. In Figure 2.5, two memory cells are displayed for a two time steps unrollment.

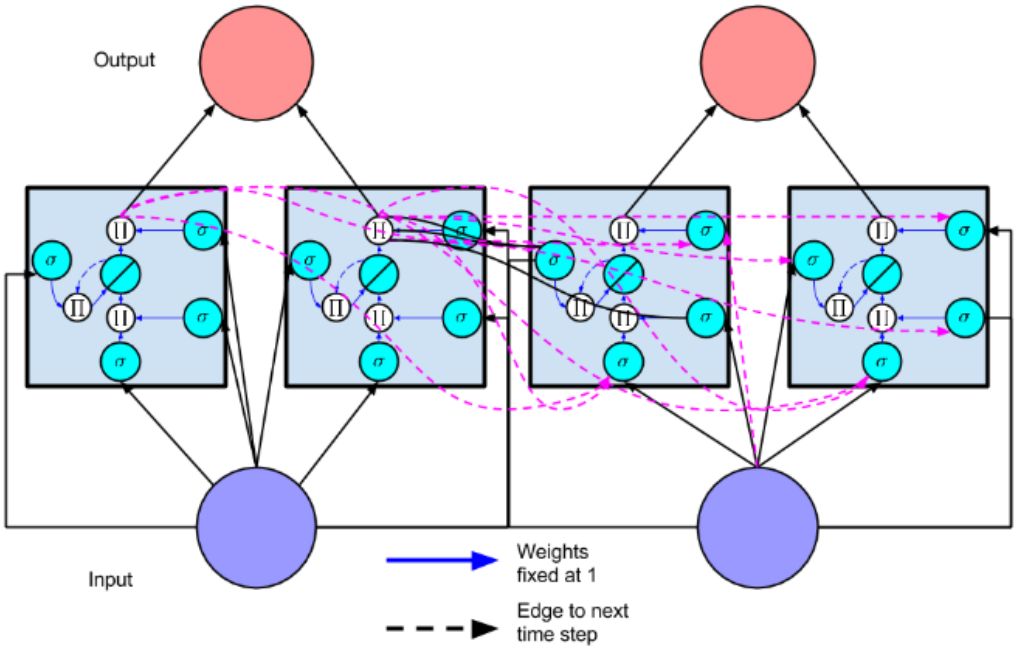

Figure 2.5: LSTM layer illustrated from [10]. The layer consists of two memory cells and is unrolled for two time steps. The purple lines indicate the connections between timestep $t-1$ and timestep t. Although it is not so clearly visualized, the internal state is also connected between time step $t-1$ and t.

Chapter 3

Data

The data used for this thesis was obtained through the Swiss bank Dukascopy's historical data feed [15]. Considering the source, the data is considered reliable. The original data consisted of minutely closing bid prices between 10.05.2016 00:00:00.000 and 10.05.2019 22:59:00.000. The data was downsampled to 10 minute averages for two reasons. Fist, it was done to reduce the number of data points. Secondly, it was done to smoothen the data and remove some of the very short term stochastic behavior. In Figure 3.1 the downsampled data is visualized.

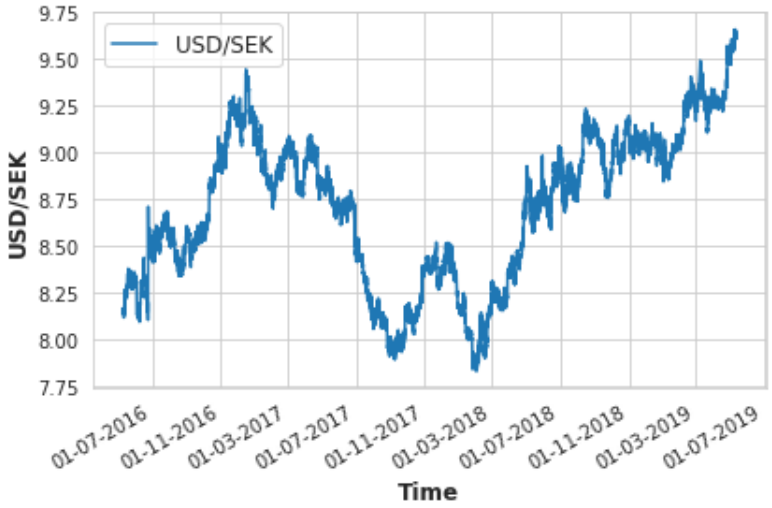

Figure 3.1: All the downsampled USD/SEK data. One observation is a ten minute average.

Chapter 4

Implementing the FNN and Results

One-step prediction is provided for the FNN. All results reported have been obtained using the last epoch's parameters. Hence there has been no cherry picking of best performing model.

4.1 One-step prediction

4.1.1 Implementation

When applying the FNN to temporal sequences, such as time series, the problem of how to processes samples which are not independent arises. There exists no mechanism in the FNN architecture which enables it to remember previous outputs. One way to mitigate this is by using the *sliding window method* which is also referred to as the *lag method*. Previously observed values are lagged and used as inputs while the next upcoming value in the time series is the output variable. By using the theory covered in Chapter 2, such an FNN was constructed to predict the next ten minute average closing bid of the USD/SEK.

Thus the first decision to be made was how many previous observations to be used as inputs. After experimentation it was found that the model was far more dependent on the amount of data it was able to train on compared to the number of lags. Thus, with the intent of keeping variables to optimize at at low level, the last hour's observed values were set to be used as inputs. Considering the 10 minute interval between observations, the data was lagged 1 to 6 observations. Table 4.2 illustrates the lagging of data for time $t = 7, 8, 9, \ldots, T$. The first observed output value is at time $t = 7$ since previous observations lack inputs.

Before preprocessing the data, the network architecture had to be decided upon. The network is identical to the one in Figure 2.1 but with the input layer in $\mathbb{R}^6$, the hidden layer in $\mathbb{R}^8$ and the output layer in $\mathbb{R}$. Considering one value was to be predicted, it was natural to let the network return one value. The size of the hidden layer was determined through trial and error. Considering the advantages of the ReLU function elaborated on in Section 2.4, both the hidden layer and the output layer used it as activation function. The program written has been summarized in Algorithm 1 on page 23.

Inputs				Output
y_{t-6}	y_{t-5}	$\cdots$	y_{t-1}	y_t
y_{t-5}	y_{t-4}	$\cdots$	y_t	y_{t+1}
$\vdots$	$\vdots$	$\ldots$	$\vdots$	$\vdots$
y_{T-6}	y_{T-5}	$\cdots$	y_{T-1}	y_T

Table 4.1: Lagging of data for the FNN. The last six previously observed values were used as input.

4.1.2 Results

At first, the prediction results look promising. In Figure 4.2, we can observe that the FNN is able to follow the exchange rate closely. Furthemore, in Figure 4.3 we can see that the network converges. The test loss converges with the train loss and does not increase, indicating there has been no overfitting.

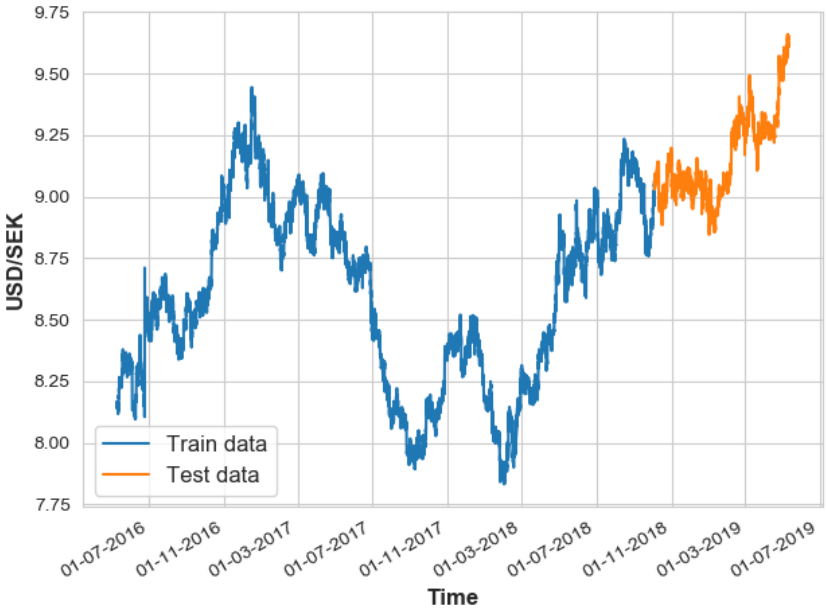

Figure 4.1: Splitting of data for the FNN. Blue indicates training data (80%) while orange indicates the test data (20%).

Alas, when the FNN performance results were benchmarked with the naïve predictor, $\hat{y}_t = y_{t-1}, \;\; t = 7, 8, 9, \ldots, T$ it failed to outperform. The performance metrics can be reviewed in Table 4.2 and the naïve predictor outperforms the FNN in all. Please note that error standard deviation will be abbreviated with error std throughout this thesis.

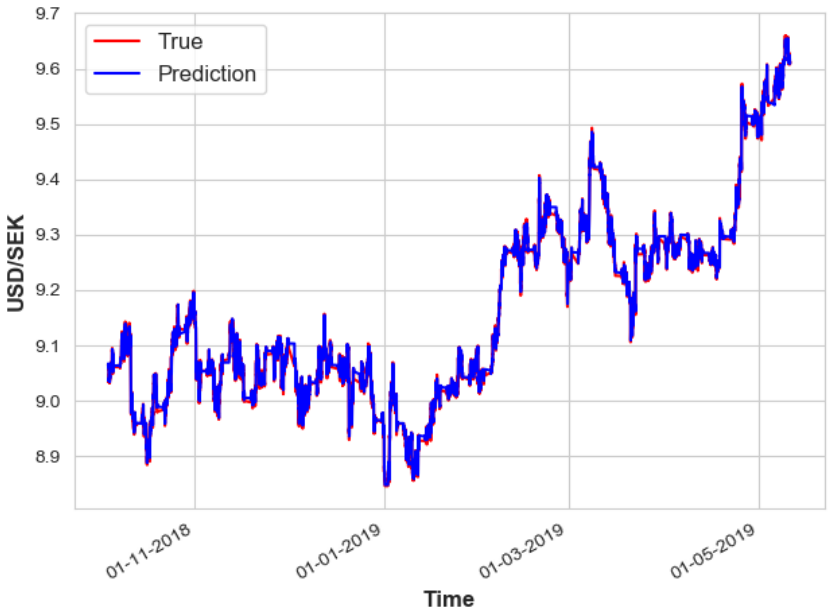

Figure 4.2: FNN prediction results on test data. The FNN follows the true USD/SEK exchange rate closely.

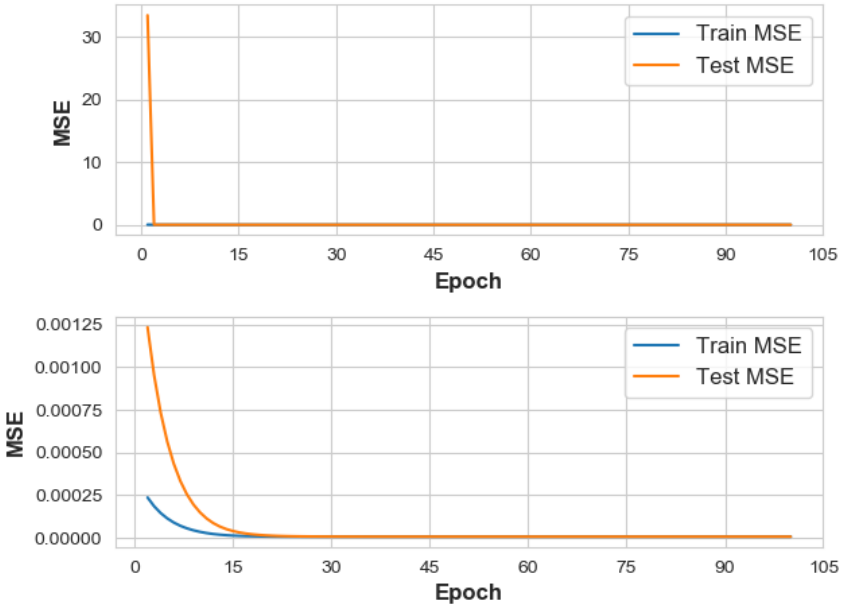

Figure 4.3: FNN train and test loss results. Top figure starts with epoch one. Bottom figure starts with epoch two. Note that test loss is greater than train loss.

Algorithm 1: Algorithm for the FNN

Result: This program predicts the USD/SEK exchange rate using the FNN. It implements feedforward- and backpropagation as covered in Section 2.2

Load and preprocess data with (0,1) transformation;
Split into train set (80%) and test set (20%);
Initialize weights $\mathbf{W}^{(1)}_{6x8}, \mathbf{W}^{(2)}_{8x1} \sim U(0,1)$;
Initialize biases $\mathbf{b}^{(1)}_{8x1}, \mathbf{b}^{(2)}_{1x1} \sim U(0,1)$;
Set eta = 0.0001;
for *100 epochs* **do**
  Test;
  for $p = 1, \ldots, N$ *in test data* **do**
    Feedforward propagation;
    Set $o_1 = x_p$;
    $net_2 = w_1^T o_1 + b_1$;
    $o_2 = ReLU(net_2)$;
    $net_3 = w_2^T o_2 + b_2$;
    $o_3 = ReLU(net_3)$;
    Set $\hat{y}_p = o_3$;
    $e_p = \hat{y}_p - y_p$;
    return $\hat{y}_p$;
    return e_p^2;
  end
  Train;
  for $p = 1, \ldots, N$ *in training data* **do**
    Feedforward propagation;
    Set $o_1 = x_p$;
    $net_2 = w_1^T o_1 + b_1$;
    $o_2 = ReLU(net_2)$;
    $net_3 = w_2^T o_2 + b_2$;
    $o_3 = ReLU(net_3)$;
    Set $\hat{y}_p = o_3$;
    $e_p = \hat{y}_p - y_p$;
    return $\hat{y}_p$;
    return e_p^2;
    Backpropagation;
    $d_3 = -e_p \cdot ReLU'(net_3)$;
    $b_2 = b_2 + eta \cdot d_3$;
    $w_2 = w_2 + eta \cdot d_3 \cdot o_2$;
    $d_2 = d_3 \cdot w_2 \cdot ReLU'(net_2)$;
    $b_1 = b_1 + eta \cdot d_2$;
    $w_1 = w_1 + eta \cdot o_1 d_2^T$
  end
  return MSE of train data;
  return MSE of test data;
end

Performance metric	FNN	Naive
Error mean	8.21e-04	2.41e-05
Error std	7.15e-03	3.58e-03
Error MSE	5.18e-05	1.28e-05
Error RMSE	7.19e-03	3.58e-03

Table 4.2: Performance metrics for the FNN and the naïve predictor. The naïve predictor outperforms the FNN in all metrics.

Chapter 5

Implementing the LSTM and Results

LSTMs have been applied with great success in various tasks such as speech recognition, language modeling and translation to mention a few [4]. Due to their integrated long term memory capabilities, LSTMs are able to exploit structure in long sequences of data where observations in the beginning effects the data at the end. Translated to this thesis, the underlying idea would be that the there exists a long-term pattern which the network can detect and approximate as a function.
Three models (numbered one to three) and their predictions are included. First, two LSTMs trained to predict one time step ahead will be presented, which differ in the lengths of input sequences, i.e., how far they are allowed to look back for their prediction. Thereafter a multi-step prediction is presented, where the LSTM's task is to predict the exchange rate further into the future.
All results reported have been obtained using the last epoch's parameters. Hence there has been no cherry picking of best performing model.
Due to local hardware limitations, implementation was done in Google Colaboratory which enables the use of a NVIDIA Tesla K80 GPU.

5.1 One-step Prediction

5.1.1 Model 1: Implementation

The network architecture was the first matter to be considered. There are more hyperparameters to be tuned in an LSTM which poses as a challenge. The LSTM is capable of remebering long sequences of data. However, to observe performance using data close to output, the last 30 observations were initially used. In terms of time, this constituted ten minute averages during a 300 minute interval. Thereafter several configurations of network components and parameters were tried out. The key aspects of the one found most effective are listed below as bullet points along with pseudo code.

- **Output layer activation function**: One common practice when predicting values is to use no activation function in the output layer. This allows the network to have the entire real line as its output.

- **Preprocessing**: Considering there was no activation function in the output layer, the log & zero mean transformation as presented in Section 2.5 could be applied.
- **Loss function**: The MAE from Section 2.6.2 was found to be the most effective loss metric.
- **LSTM layers**: LSTM layers must not be deployed alone but can also be stacked. Instead of only returning the final output in the first LSTM layer, the output of the entire layer $\mathbf{h}^{(t)}$ (see Figure 2.5), is used as input in the next LSTM layer. The final setup deployed two stacked LSTM layers.

Algorithm 2: Algorithm for LSTM network

Result: This program predicts the USD/SEK exchange rate using the LSTM neural network implemented in Keras with TensorFlow backend.

Load and preprocess data with log & zero mean transformation;
Split into train set (80%) and test set (20%);
Declare network;
Two stacked LSTM layers with 64 memory cells each;
One dense layer with linear activation;
for *100 epochs* **do**
    Train and test network by applying;
    AMSGrad optimizer;
    512 batch size;
    MAE loss;
end

5.1.2 Model 1: Results

In comparison to the FNN, the results have improved. The network outperforms the naïve predictor, $\hat{y}_t = y_{t-1},\ t = 31, 32, 33, \ldots, T$, in all metrics except the error mean as can be reviewed in Table 5.1. It is not by a large margin however.

Performance metric	LSTM	Naïve
Error mean	2.91e-04	-2.49e-05
Error std	3.52e-03	3.58e-03
Error MAE	2.24e-03	2.24e-03
Error MSE	1.25e-05	1.28e-05
Error RMSE	3.53e-03	3.58e-03

Table 5.1: Performance metrics for the LSTM model 1 and the naïve predictor. The LSTM outperforms the naïve predictor in all metrics except the error mean. MAE results are equal. Model 1's task was to predict the next observation's value. In other words, it did one-step prediction. The last 30 observations were used as input.

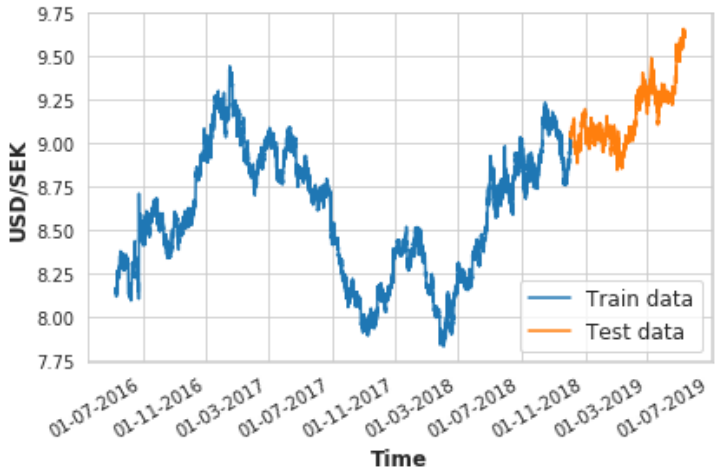

Figure 5.1: Splitting of data for the LSTM model 1. Blue indicates training data (80%) while orange indicates the test data (20%).

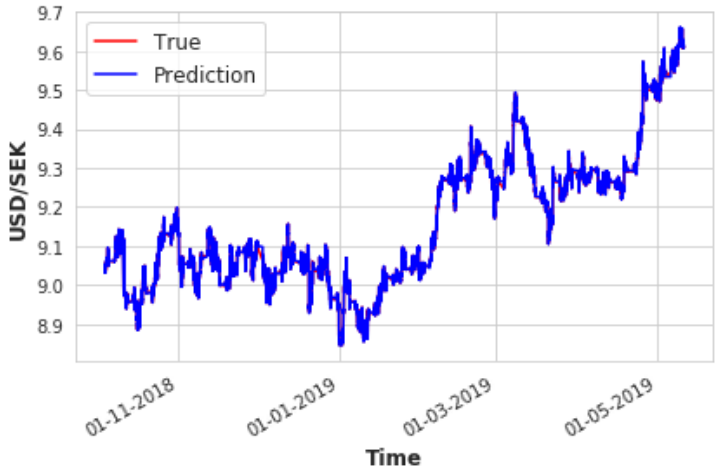

Figure 5.2: LSTM model 1 prediction results on test data. The LSTM follows the true USD/SEK exchange rate closely.

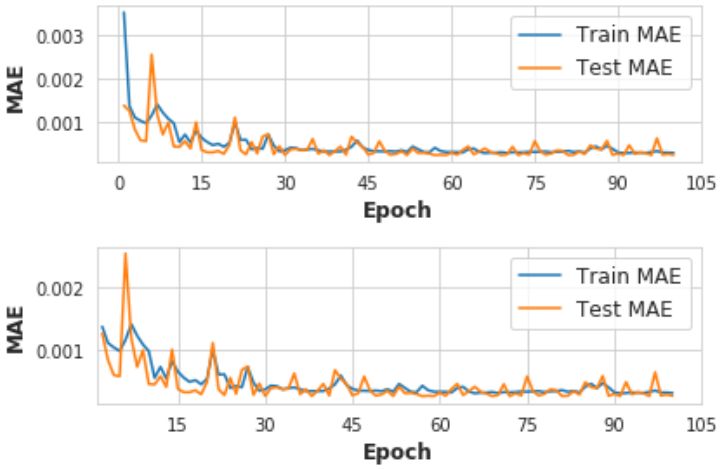

Figure 5.3: LSTM model 1 train and test loss results. Top figure starts with epoch one. Bottom figure starts with epoch two. There is a peak at approximately epoch six for the test loss. Model does not appear to converge completely since the losses are not flat in the final epochs. Keras computes the test loss at the end of each epoch, while the training loss is computed batchwise and averaged at the end of each epoch.

5.1.3 Model 2: Implementation

Considering the results obtained using the previous 30 observations as inputs, the idea of expanding the horizon looking back was worth exploring. Instead of only using the previous 30 observations, the last 2016 were used, constituting two weeks of 10 minute averages for the LSTM to take into consideration when predicting the next value. The algorithm was the same as Algorithm 2 but with epochs set to 50 instead of 100. A split into 80% training and 20% test data was perfomed as above.

5.1.4 Model 2: Results

Inspecting the results in Table 5.2, the LSTM fails to outperform the naïve predictor $\hat{y}_t = y_{t-1}, \;\; t = 2017, 2018, 2019, \ldots, T$ in all metrics except the error std.

Performance metric	LSTM	Naïve
Error mean	1.79e-03	-2.82e-05
Error std	3.53e-03	3.58e-03
Error MAE	2.82e-03	2.23e-03
Error MSE	1.57e-05	1.28e-05
Error RMSE	3.96e-03	3.58e-03

Table 5.2: Performance metrics for the LSTM model 2 and the naïve predictor. The LSTM fails to outperform the naïve predictor in all metrics, except in error std. Model 2's task was to predict the next observation's value. In other words, it did one-step prediction. The last 2016 observations were used as input.

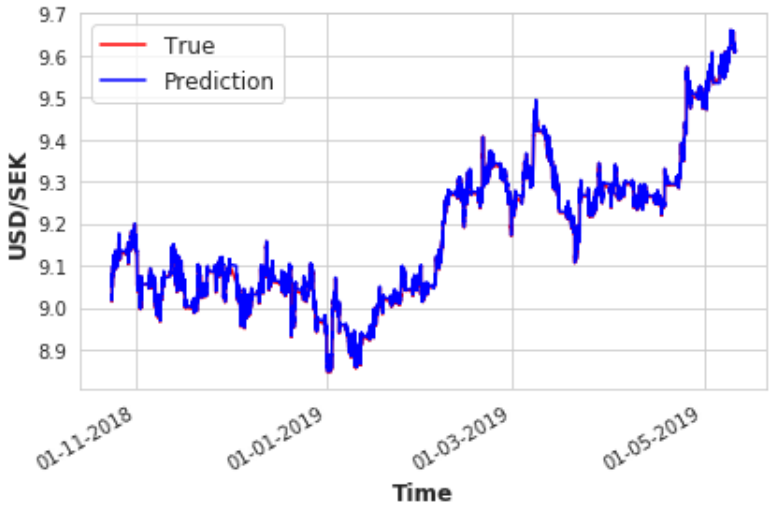

Figure 5.4: LSTM model 2 prediction results on test data. The LSTM follows the true USD/SEK exchange rate closely.

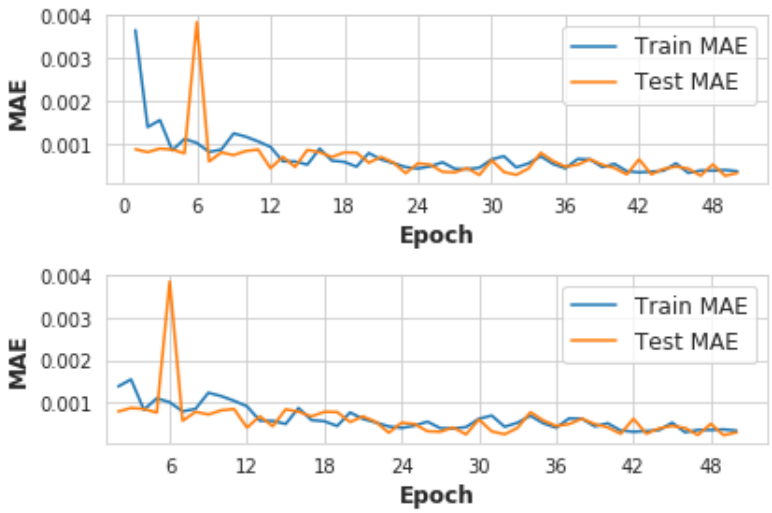

Figure 5.5: LSTM model 2 train and test loss results. Top figure starts with epoch one. Bottom figure starts with epoch two. There is a peak at approximately epoch six for the test loss. Model does not appear to converge completely since the losses are not flat in the final epochs. Keras computes the test loss at the end of each epoch, while the training loss is computed batchwise and averaged at the end of each epoch.

5.2 Multi-step Prediction

5.2.1 Model 3: Implementation

Next, it was experimented whether a multi-step prediction would outperform the naïve predictor, as the naïve predictor is considerably worse in predicting multiple time steps ahead. Instead of predicting the next step, the 12'th step was to be predicted. In terms of data, it would constitute the 10 minute average in two hours. The set up was the same as in algorithm 2 and used the last 2016 observations as input. Epochs were set to 50 instead of 100. A split into 80% training and 20% test data was perfomed as above.

5.2.2 Model 3: Results

As not to compare apples and oranges, the naïve predictor was redefined as $\hat{y}_t = y_{t-12},\ t = 2017, 2018, 2019, \ldots, T$. Not unexpectedly the error metrics have gone up for the naïve predictor. And judging from Table 5.3, it appears as if the LSTM is slightly underperforming the naïve predictor. In error std and RMSE the LSTM has achieved the same results as the naïve predictor. In all other metrics, the naïve predictor outperforms the LSTM by a very small margin.

Performance metric	LSTM	Naïve
Error mean	-1.29e-03	-3.44e-04
Error std	1.42e-02	1.42e-02
Eroor MAE	9.38e-03	9.32e-03
Error MSE	2.03e-04	2.01e-04
Error RMSE	1.42e-02	1.42e-02

Table 5.3: Performance metrics for the LSTM model 3 and the naïve predictor. It performs as well as the naïve in error std and error RMSE. It fails to outperform in the other metrics. It is however only by a small margin. Model 3's task was to predict the next 12'th observation's value. In other words, it did twelve-step prediction. The last 2016 observations were used as input.

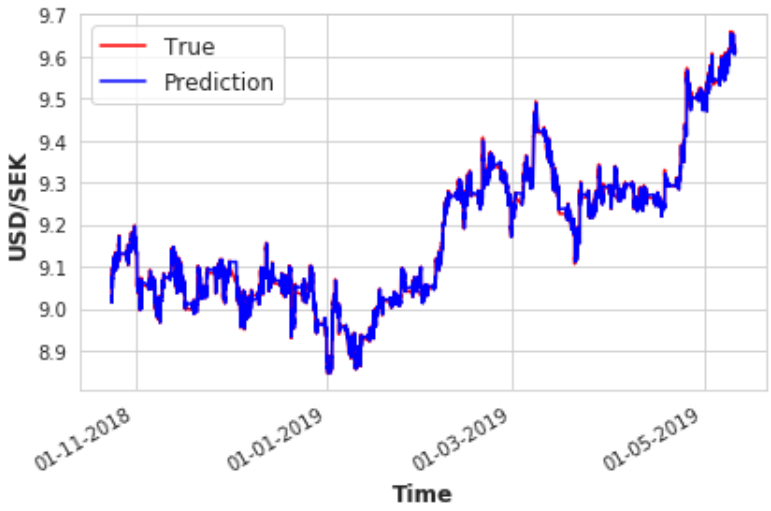

Figure 5.6: LSTM model 3 prediction results on test data. The LSTM follows the true USD/SEK exchange rate closely.

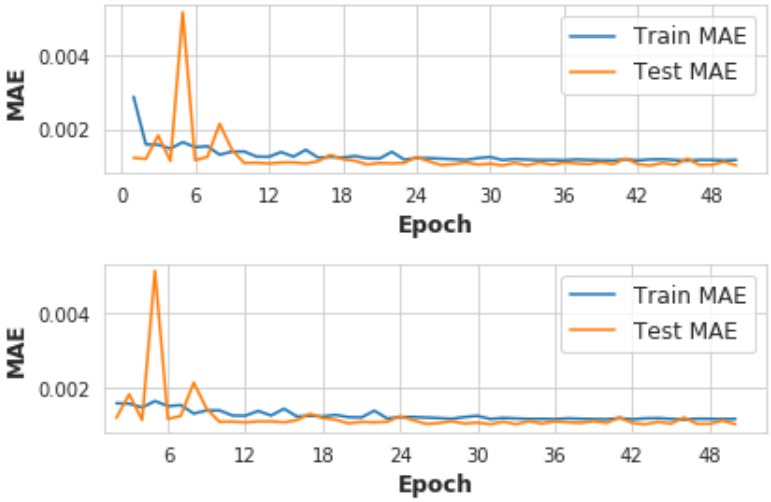

Figure 5.7: LSTM model 3 train and test loss results. Top figure starts with epoch one. Bottom figure starts with epoch two. There is a peak at approximately epoch six for the test loss. Model does appear to converge completely since the losses are flat in the final epochs. Keras computes the test loss at the end of each epoch, while the training loss is computed batchwise and averaged at the end of each epoch.

Chapter 6

Discussion

6.1 Results Discussion

Considering the superiority of the LSTM to the FNN, the LSTM results will be discussed in more detail.

6.1.1 Model 1

This model outperformed the naïve predictor in all metrics, except in error mean, which can be seen in Table 5.1. But the case is not clear cut. When inspecting the the behavior of train and test loss in Figure 5.3 three things are noteworthy.

First, the train and test loss both drop sharply after just a few epochs. As to combat this, different learning rates were applied to the AMSGrad optimizer as well as trying other optimizers. But these measures did not change the outcome. Another cause for concern was the very small errors from the loss metric. It could be a problem when updating the parameters as the small error will make the changes very small.

Secondly, the relationship between the train and the test loss is not conventional. Usually, test loss > train loss as in the FNN's case in Figure 4.3. The thought of the network being able to perform better on data on which it has not trained is confusing. For this there could nevertheless be a mechanical explanation. It stems from how Keras computes the losses from training and testing after each epoch. In the Keras documentation it says "...the training loss is the average of the losses over each batch of training data. Because your model is changing over time, the loss over the first batches of an epoch is generally higher than over the last batches. On the other hand, the testing loss for an epoch is computed using the model as it is at the end of the epoch, resulting in a lower loss" [9].

Thirdly, when looking at the train and test loss towards the later epochs, there is some convergence. But when inspected more closely, test loss was not strictly monotonically decreasing, so the result obtained in Table 5.1 could also have been higher for the LSTM.

6.1.2 Model 2

The results in Table 5.2 show that LSTM model 2 fails to outperform the naïve predictor in all metrics except the error std. When inspecting the train and test loss

in Figure 5.5, much of the erratic behavior in the first epochs are gone. Meanwhile, from looking at the behavior during the final epochs, it seems as if the network has not yet converged. Hence it would be advisable to run the model for more epochs.

6.1.3 Model 3

Formally, the LSTM model 3 fails to outperform the naïve predictor in all metrics except error std as can be seen in Table 5.3. But the margin is very small. When reviewing the final epochs in Figure 5.7, the network appears to have converged. In addition, the lack of erratic behavior in the first epochs is very similar to the result in LSTM model 2. Another similarity to LSTM model 2 is the spike in test loss around epoch six. And if watched closely, there also appears to be a spike around epoch six in LSTM model 1. The reason for this behavior is unknown. But it is suspected to come from how Keras initializes the model parameters. All three LSTM models are similar. And when the gradient descent is embarked upon, it could be that all models travel along a similar path where they all at approximately epoch 6 make the same error. Due to Keras averaging train losses during an epoch, the spike is barely visible in the train loss.

6.2 Conclusions

It would be tempting to conclude that the LSTM outperforms the FNN. Although the data analyzed is not 1-to-1 for the FNN and LSTM model 1 and 2 they could be compared since they all performed one-step predictions. And the two LSTM models outperformed the FNN in all metrics. Not to be forgotten however, the LSTM models have been more emphasized throughout this thesis. The FNN constructed is a much smaller network compared to LSTM model 1 or 2. Hence it cannot be excluded that the outperformance comes from network size and success in training.

- **Conclusion 1**: The LSTM models learn to become the naïve predictor with some modification.

This thesis concludes that the LSTM models learn to become the naïve predictor with some modification. It is supported by the fact that error metrics are very similar for all three models and the naïve predictor. All the LSTM models fail to outperform in the error mean. But this must not be so bad. The error mean is sensitive to sign changes in its summation. If the naïve predictor tends to over- and undershoot the true value more than the LSTM models, then the error mean will be smaller as positive and negatives values cancel each other. Perhaps more interesting is that the LSTM models outperform, or are equal, to the the naïve predictor in error std. It points to that the LSTM models' errors are less spread out than the naïve errors.

During a long time while writing this thesis, the MSE and RMSE were used as loss metrics. No LSTM model was observed outperforming the naïve predictor while using the MSE. One cause for concern which arouse while doing so was the smallness of the loss. It was suspected such a small error provided little information when updating the gradients. This led to the introduction of the RMSE which improved the results. While using the RMSE, LSTM model 1 was observed outperforming the naïve predictor. It was however hard to justify penalizing the LSTM's predictions

the way MSE and RMSE do (see Section 2.6). This led to the final change and introduction of the MAE. As can be observed in Table 5.1, LSTM model 1 performs as good as the naïve predictor in MAE while outperforming in both MSE and RMSE.

- **Conclusion 2**: Longer lookback does not necessarily improve the performance of the LSTM.

This thesis also concludes that there appears to be no great improvement for the LSTM when the lookback horizon is extended. The LSTM model 2 struggles to find any existing structure on which to generalize. It is supported by the fact that the results in Table 5.2 did not improve greatly when the input lookback was increased from 30 to 2016. The result might not be final however. When inspecting the losses in the final epochs in Figure 5.5 there seems to be room for further convergence. Hence it would be advised to run for more epochs to obtain more final results. In addition, it could be that a lookback of 2016 observations not was enough. If longer lookback would be applied, there could be more structure.

Another tempting conclusion to be drawn would be that for multi-step prediction, the LSTM resorts to the naïve predictor. This would be supported by the results obtained in Table 5.3 where LSTM model 3 and the naïve predictor have very similar error metrics. In error std and RMSE it is equal. In MSE and MAE its almost the same. Furthermore, when looking at Figure 5.7 there is convergence in the final epochs leaving little room for the LSTM model 3 to improve. Nevertheless, this thesis only did multi-step prediction for 12 time steps and the outcome could be different for another multi-step prediction.

Financial data is infamously difficult to work with. Considering all LSTM models struggle with outperforming the naïve predictor, it can be suggested the data is random and lack structure. Certainly, a case could be made that with another type of preprocessing, network architecture or parameter configuration, better results could perhaps be obtained. But the permutations of such models are endless. However, one limitation this thesis acknowledges is the model size of LSTM model 2 and LSTM model 3. Considering the input of 2016 time steps, it could be that a larger network would have performed better. But in deep learning, the trade-off between complexity and run-time is always present.

In the following section, some suggestions are put forward on what could be considered next or instead.

6.3 Looking Ahead

6.3.1 Multivariate LSTM and Problem Reframing

The most advanced model applied in this thesis was the univariate LSTM. A natural next step would be to apply the multivariate LSTM. Instead of only using previously observed values of an exchange rate as inputs, other parameters could be used. The data initially obtained contained OHLCV (opening, high, low, close and volume) of USD/SEK bid side. This thesis only used the closing data.

Another aspect to consider is the complexity of the problem. Although the results of this thesis indicate that the LSTM is able to outperform the naïve predictor, it still struggles to do so. Hence it could be worthwhile to reduce the complexity of the problem. Instead of predicting a real value, a two dimensional output such as "up or down" could perhaps be more successful.

6.4 Economic Theory to Improve the Results

The lack of access to other financial data should not be understated. For example, the Federal Reserve Economic Database (FRED) offers free economic data, such as LIBOR rates. A limitation however is that it is daily and only dates back a few years. More data is needed for effective deep learning. If a similar project like this thesis is carried out in a commercial setting where financial data is more abundant it could prove fruitful to review some economic theory on foreign exchange rates. A natural point of departure would be the *interest rate parity* condition.

6.4.1 Interest Rate Parity

Paul R. Krugman (a Nobel Prize awarded economist), Maurice Obstfeld and Marc J. Melitz suggest a currency should be viewed as an asset and the exchange rate as the relative price of one asset expressed in terms of another [7]. And as with any other asset, the forces of supply and demand dictate the price. Furthermore, demand is said to be driven by (1) the currency's interest rate and (2) the expected change in the exchange rate. This leads to the basic equilibrium condition, namely interest rate parity [7]. Let A and B be two different currencies and r_A and r_B the rate of interest from time t_0 until t_1 for each. The rate of interest is what determines what is earned on deposits of each currency. Denote the exchange rate at time t_0 by $E^0_{A/B}$ and the expected future exchange rate at time t_1 by $E^1_{A/B}$. The equilibrium condition can then be stated as

$$r_A = r_B + \frac{E^1_{A/B} - E^0_{A/B}}{E^0_{A/B}}. \tag{6.1}$$

In other words, regardless if one holds currency A or B at time 0, deposits it, withdraws it at time t_1 and exchanges it, the value should be the same as if the exchange was made at t_0. In theory, if the equilibrium is violated, then market forces should act upon it and remove any arbitrage opportunities. It is worthwhile mentioning that two other factors have been neglected throughout this view. Those are, the risk of the asset and its liquidity. This simplification is justified as it makes the macroeconomic analysis easier [7]. However, the question of what determines the interest rates and the expected future exchange remains unanswered. Economists have built elaborate models which list inflation, unemployment and other macroeconomic variables as explanatory variables to mention a few [7].

6.5 Biases and Other Factors

This thesis did not take into consideration any biases or limitations in its analysis. But for any commercial extension some immediate biases should be considered:

- **Look-Ahead Bias**: This thesis did transform and manipulate the data it used as test data before the analysis. It is a bias which favours the results obtained. For example, this thesis has applied the (0,1) transformation where the maximum and minimum of all the data, training as well as testing, was used. In a setting where new samples would be generated as time passes, such a transformation would be impossible as the true population maximum and minimum could still be unobserved.
- **Spread**: This thesis only looked at the bid side of the FX market. For a more complete analysis, the ask side should also be taken into account.
- **Transaction Costs and Tax effects**: In an extension, it should be considered whether the results of the model are good enough to compensate for any transaction costs. Finally, tax effects should also be considered.

Bibliography

[1] Denny Britz. *Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients*. 2015-10-08. URL: http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/ (visited on 05/08/2019).

[2] Warren S. Sarle. *Should I nonlinearly transform the data?* 2014-03-27. URL: http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-17.html (visited on 05/27/2019).

[3] Warren S. Sarle. *Should I normalize/standardize/rescale the*. 2014-03-27. URL: http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-16.html (visited on 05/27/2019).

[4] Christopher Olah. *Understanding LSTM Networks*. 2015-08-27. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (visited on 04/30/2019).

[5] Nasdaq.com. *Forex Market Overview*. no date. URL: https://www.nasdaq.com/forex/education/foreign-exchange-market-overview.aspx (visited on 04/26/2019).

[6] George Cybenko. "Approximations by superpositions of sigmoidal functions". In: *Mathematics of Control, Signals, and Systems* 2.4 (1989), pp. 303–314. DOI: 10.1007/BF02551274.

[7] Marc J Melitz Paul R. Krugman Maurice Obstfeld. *International Economics Theory & Policy 9th Global Edition*. Pearson Education Limited, 2012. ISBN: 978-0-273-75409-1.

[8] Ke-Lin Du, M. N. S. Swamy. *Neural Networks and Statistical Learning*. Springer, 2014. ISBN: 978-1-4471-5570-6.

[9] Franccois Chollet et al. *Keras*. 2015. URL: https://keras.io/getting-started/faq/#why-is-the-training-loss-much-higher-than-the-testing-loss.

[10] Charles Elkan Zachary C. Lipton John Berkowitz. "A Critical Review of Recurrent Neural Networksfor Sequence Learning". In: (2015). URL: https://arxiv.org/abs/1506.00019.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[12] *Compare the effect of different scalers on data with outliers*. 2017. URL: https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html#sphx-glr-auto-examples-preprocessing-plot-all-scaling-py (visited on 06/15/2019).

[13] Danqing Liu. *A Practical Guide to ReLU*. 2017. URL: https://medium.com/tinymind/a-practical-guide-to-relu-b83ca804f1f7.

[14] Noboru Murata Sho Sonoda. "Neural Network with Unbounded Activation Functions is Universal Approximator". In: *Applied and Computational Harmonic Analysis* 43.2 (2017), pp. 233–268. DOI: 10.1016/j.acha.2015.12.005.

[15] Dukascopy. *Historical Data Feed*. 2019. URL: https://www.dukascopy.com/swiss/english/marketwatch/historical/ (visited on 06/01/2019).

Bachelor's Theses in Mathematical Sciences 2019:K22
ISSN 1654-6229

LUNFMS-4037-2019

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/