FACULTEIT ECONOMIE EN BEDRIJFSKUNDE

COMPARING CONVOLUTIONAL AND LSTM NEURAL NETWORKS FOR INTRADAY FOREX DIRECTION PREDICTION

Aantal woorden/ Word count: 15772

Simon Serrarens

Studentennummer/ Student number : 01306253

Promotor/ Supervisor: Prof. dr. Dirk Van den Poel

Co-promotor/ Co-supervisor: Arno Liseune

Masterproef voorgedragen tot het bekomen van de graad van:

Master's Dissertation submitted to obtain the degree of:

Master of Science in Business Engineering

Academiejaar/ Academic year: 2017 – 2018

PERMISSION

Ondergetekende verklaart dat de inhoud van deze masterproef mag geraadpleegd en/of gereproduceerd worden, mits bronvermelding.
I declare that the content of this Master's Dissertation may be consulted and/or reproduced, provided that the source is referenced.

Naam student/name student : Serrarens Simon

Handtekening/signature

Serrarens Simon

Foreword

As this thesis represents the culmination of my studies as a business engineer, it feels appropriate to look back and reflect. On what I have learned, but also on how I got there and who helped me along the way. My studies have been an amazing experience, and looking back on it now I can only be grateful.

First, I would like to thank professor Dirk Van den Poel, for accommodating this thesis, as well as providing feedback on results, and giving ideas for alternative approaches. I also want to thank Arno Liseune, for helping with several issues and guiding me in defining my research topic.

I also want to thank Pauline, for listening and answering patiently to more than a couple of questions, as well as giving suggestions for my research.

In the last five years we have been through quite some group assignments. I would like to thank Jens, Nils, Steven and Willem for hard work, long nights, interesting discussions on the different topics, but most of all for the life-long friendships.

I am also very grateful to my wonderful girlfriend, Nella, for always being interested in what I'm doing, thoroughly reading my work, and being so supportive.

Last but not least, I would like to thank my mother. For providing me with the opportunity to study and develop myself, for all the support throughout the years and helping me to achieve my dreams.

Table of contents

List of used abbreviations

ANN	Artificial neural network
CLSTM	Convolutional long short-term memory
CNN	Convolutional neural network
EMH	Efficient market hypothesis
FFNN	Feedforward neural network
FOREX	Foreign exchange market
HAR	Human activity recognition
LSTM	Long short-term memory
MLP	Multilayer perceptron
ReLU	Rectified linear unit
RUL	Remaining useful life estimation

List of tables

List of figures

1. Introduction

1.1 Predicting the foreign exchange market

Today, the world's financial markets have an enormous impact on modern economies. In the different markets, trillions of euros worth of financial products are traded on a daily basis. The foreign exchange market (FOREX) is no exception to this; it is a market that averaged $5.09 trillion per day in April 2016, according to the Bank for International Settlements (Triennial central bank survey, 2016). All these transactions shape the exchange rates of currencies, which fluctuate over time. Given the high dollar volume in daily trades, being able to predict the changes in these rates opens up possibilities for tremendous profits.

For a time, it was generally considered impossible to predict rates in the different economic markets, such as stock prices and indices, and exchange rates. This was articulated by Fama (1970) in his Efficient Market Hypothesis (EMH). He claimed that there was plenty of evidence that the financial markets are at least semi-strong, meaning that the prices fully reflect all publicly available information, at all times, and that thus no excessive gains can be made when new information comes to light, as markets immediately correct themselves. Malkiel (1973), goes further still, stating that in the short-term, no predictions can be made about the stock market, and that all prices fluctuate as if they were random. His theory builds on the EMH, and is known as The Random Walk Down Wall Street. These hypotheses have not stopped investors, however, from trying to predict these rates, and make a profit out of their predictions. In 2012, an estimated 85% of the traded volume of US stock market was done based on algorithms (Glantz & Kissell, 2013).

By the early 21st century it was accepted that financial rates are at least somewhat predictable (Malkiel, 2003). It appears that even though they are rare, sometimes bubbles can exist, causing a short-term and limited possibility for investors to make above-average profits.

1.2 Financial time series forecasting

Exchange rates, stock prices and indices are, in fact, time series. Time series forecasting is a broad scientific topic that has received a lot of attention and seen an evolution in the last decades. Time

series are omnipresent, and researchers are looking to predict them in different fields, e.g. speech recognition, human activity recognition, remaining useful life estimation, financial rates, etc.

De Gooijer & Hyndman (2006) believe that Yule was one of the first researchers to introduce the idea that time series are stochastic, by posing that all time-series can be thought of as the realization of a stochastic process, in 1927. This has, in turn, lead to a number of early forecasting techniques for these time series, such as exponential smoothing and moving averages. In 1970, Box and Jenkins integrated the existing work and created the first methodology for identifying and analysing discrete time-series (Box & Jenkins, 1970). The Box-Jenkins methodology could be used to build autoregressive integrated moving average (ARIMA) models. The use of these models further increased with the advent of modern computing power (De Gooijer et al., 2006), which in turn lead to an increasing interest in time-series forecasting.

While ARIMA models have been immensely popular, and were often used as benchmark models in time-series forecasting, their assumption that the time-series being forecasted are stationary and linear does not hold for financial time series (Kamruzzaman & Sarker, 2003). Machine learning techniques, however, do not have this restriction. Several machine learning algorithms have proven to be useful for financial time series forecasting to greater or lesser extent. Ballings, Van den Poel, Hespeels & Gryp (2015) provide an overview of publications that have applied different machine learning techniques to the financial time series forecasting problem, and provide a benchmark themselves, stressing the importance of ensemble models. One of the techniques used in their study are feedforward neural networks (FFNN). These networks are part of a larger category of techniques, called artificial neural networks (ANN). Even though early ANN methods initially proved to be unsuccessful in predicting financial time series (White, 1988), a lot of research has been done ever since. Atsalakis & Valavanis (2009) published an overview of over 100 articles using some form of neural network for stock market forecasting, finding that the majority of these techniques outperform the more conventional forecasting techniques, having higher accuracy and yielding better results as trading systems.

1.3 Deep learning for financial time series forecasting

1.3.1 Long short-term memory networks

In this thesis we will explore a sub-area of artificial neural networks, the so-called deep neural networks (DNN). A neural network is considered to be deep if it has at least 2 hidden layers. The idea of deep neural networks was abandoned during the 90's, as the deep learning community believed that the networks would always get stuck in local minima, a sub-optimal solution. LeCun, Bengio & Hinton (2015), however, argue that in reality this actually occurs very rarely, and that most deep learning networks tend to be very robust. The advantage of these networks is that because of their several layers, they have the capacity of learning complex, non-linear relationships, the kind of relationships we find in financial time-series forecasting, and have effective learning algorithms (Armano, Marchesi & Murru, 2003). Near the end of the 20th century, interest in deep learning took off again, with several ground-breaking publications.

One of these publications included the introduction of the Long short-term memory (LSTM) network (Hochreiter & Schmidhuber, 1997). LSTM networks were proposed as a solution for the so-called vanishing gradient problem that other recurrent neural networks (RNN) at the time experienced. In this phenomenon, error signals that flow backward through time in the network tend to either vanish or blow up exponentially, depending on the size of the network weights. This leads to network training either taking too long, or rendering training infeasible altogether (Hochreiter, 1998). According to Hochreiter et al. (1997), LSTM networks are a solution to this problem as they use input and output gated units, controlling what inputs get into the memory unit, preventing it from taking up unnecessary information. The output gates control what information leaves the unit, and is sent to units in consecutive layers. This architecture, including the forget gates that were later introduced by Gers, Schmidhuber & Cummins (1999), is now considered the standard LSTM unit, which builds up an LSTM network. In a later section we will delve deeper into the LSTM architecture. LSTM networks have been successfully applied to a wide variety of time-series prediction problems and temporal classification problems, including speech recognition (Graves, Mohamed & Hinton, 2013), remaining useful life estimation (Yuan, Wu & Lin, 2016), phenotyping clinical time series (Lipton, Kale & Wetzell, 2015) and traffic speed prediction (Ma, Tao, Wang, Yu & Wang, 2015); as well as other fields such as music composition (Eck & Schmidhuber, 2002). Application of LSTM networks for financial time series include stock index

movement prediction (Di Persio & Honchar, 2016; Nelson, Pereira & de Oliveira, 2017) and stock returns prediction (Chen, Zhou & Dai, 2015).

1.3.2 Convolutional neural networks

Similar to LSTM networks, convolutional neural networks (CNN) have also seen a recent upsurge in use and applications. At the end of the 20th century, LeCun, Bottou, Bengio & Haffner (1998) proposed that with recent computer technology and improvements in the field of machine learning, it is now possible to achieve much higher accuracy in several classification problems, such as handwriting recognition and speech recognition. In their paper, they focus especially on convolutional neural networks as the technique that outperforms all others, such as regular feedforward neural networks (FFNN). Until that time, most pattern recognition systems relied on a combination of automatic learning and hand-crafted input features. CNNs are different from other machine learning techniques in two ways. On the one hand they essentially consist of a feature extractor, transforming raw input data into high-level features as the data passes through several convolutional and aggregating (pooling) layers, and on the other hand they also act as a classifier, as convolutional neural networks usually possess several feedforward layers. Additionally, CNNs are easier to train as they will have less connections than a FFNN designed for the same problem, because in a FFNN all units are traditionally connected to all other units in the preceding and following layer, whereas units in a convolutional layer are restricted to their receptive fields (LeCun & Bengio, 1995). This makes these networks less susceptible to overfitting when there is little data. While LeCun et al. (2015) state that for sequential data RNNs are often more suitable, CNN have been applied successfully for time series classification in a variety of areas, such as human activity recognition (Zeng, Nguyen, Yu, Mengshoel, Zhu, Wu & Zhang, 2014; Yang, Nguyen, San, Li & Krishnaswamy, 2015), remaining useful life estimation (Babu, Zhao & Li, 2016), as well as financial time series forecasting, such as stock price direction prediction (Gunduz, Yaslan & Cataltepe, 2017) and exchange rate direction prediction (Di Persio et al., 2016; Galeshchuk & Mukherjee, 2017).

1.4 Conducted research and hypotheses

Given the previous success of these methods, it seems interesting to know how well they compare for financial time series prediction. In this thesis we will make a comparison between LSTM and

convolutional neural networks, for their accuracy in predicting whether exchange rates will go up or down, based on historical data and technical indicators as input, on a per-minute basis. A choice was made to predict the direction of these rates rather than the actual values as it has been argued that it is more valuable for an investor to know whether a rate will go up or down, rather than to what extent (Chao, Shen & Zhao, 2011). While this itself has already been done before, we will also attempt to create a hybrid model combining the two. In our hybrid model we combine the typical convolutional and pooling layers of CNN, and instead of then adding several feedforward layers, the output of the first layers will be used as input for an LSTM network. Such hybrid models have already successfully been applied in a variety of fields: remaining useful life (RUL) estimation (Zhao, Yan, Wang & Mao, 2017) and human activity recognition (HAR) (Ordóñez & Roggen, 2016). Vargas, de Lima & Evsukoff (2017) created a similar network, in which they used an LSTM network for technical analysis on the one hand, and a convolutional network for textual data from financial news articles on the other, combining both models in one overarching LSTM network. They use the model to predict whether stock market indices will go up or down.

We hypothesise that this hybrid model will combine the best aspects of both underlying models: On the one hand the convolutional layers will function as a feature extractor, a task for which they have been proven to be highly suitable (LeCun et al., 2015). On the other hand, the LSTM network, which is usually heavily dependent on the quality of manually engineered features (Zhao et al., 2017), will take these input features, and, being able to truly capture long-term dependencies, make accurate predictions. Hence, we hypothesise that this combined model will outperform both the CNN and LSTM network. As of this point, we will refer to the hybrid model as CLSTM network, as done by Zhao et al. (2017). As is often done, we will also build a feedforward neural network, in order to provide a more complete comparison. While we start out by building and optimising the different deep learning models for the EUR/USD currency pair, we also apply them to other currency pairs. In the making of the different models, for the different currencies, some interesting and unexpected results were obtained. After discussing the main results (section 4), we will go deeper into some ex posts tests we performed (section 5), comparing MLP and LSTM network performance as a function of the activation function, comparing CNN and CLSTM robustness, and investigating the behaviour of our models when applied to a test set consisting of data of another currency pair than EUR/USD dollar. We do this both for models trained on said other currency pair

(i.e. GBP/USD and USD/JPY), as well as for models trained on the EUR/USD currency pair, and then tested on these other pairs.

In doing so, this thesis will contribute to the existing literature in several ways. First, to the best of our knowledge, this is the first work to compare CNN, LSTM and MLP networks for FOREX direction prediction based on technical indicators, using out-of-sample data. Second, it is an attempt at combing convolutional and LSTM networks for FOREX direction prediction. Finally, we assess the generalisability of our models by training them on one currency pair, and testing them on another. We hypothesise that such a test will show that the models do indeed learn the patterns in currency data to make predictions, rather than learning the data itself. To the best of our knowledge, this has not been investigated before.

All code for this thesis is written in Python, making use of Google's TensorFlow library for deep learning. The code is added in the appendix, as well as a manual on how it is structured. It needs to be said that during training of the models other users also made use of the server, in fluctuating extent. The reported training times should be interpreted as order of magnitude values rather than fixed values for training, and used more on a comparative basis, ceteris paribus.

The rest of this thesis is structured as follows: first we will provide an overview of literature in which one or more deep learning methods were applied in financial time series forecasting, and briefly discuss their results (section 2). Next, in section 3, we will go over the inputs used for the tests in this thesis. In section 4 we will detail the different deep learning methods, giving a general overview of how they work and what different hyperparameters exist, as well as present several of the tested methods and the achieved predictive accuracy of those methods. In section 5, we present the results of the ex post tests that we performed, given the performance of our models. We will also perform some tests of statistical significance, to prove that our models do really perform better than random. Finally, in section 6, we will provide a conclusion and discuss the limitations of this thesis.

2. Related work

In this section we will give a short overview of some of the publications we consulted that specifically cover one or more deep learning methods for financial time series forecasting. We used these publications to get an impression of what constitutes relevant research, to see what relevant deep learning methods for our problem exist, as well as their parameters, and to compare results. This overview is not intended to be an exhaustive summary of the literature on deep learning for financial time series forecasting. We present the overview in table 2.1 on the next page.

The potential of deep learning for financial time series forecasting is apparent from this table: the best method found in each publication, as indicated in bold, is always a deep learning method. It does appear, however, that there are some differences in accuracy when predicting whether a price or rate will go up or down. We believe that there two main reasons for this. First, some of these articles do not use out-of-sample data. Using out-of-sample data means that the data used for training should cover a similar period as the data used for validation and testing. In our case, for example, the training data covers a four-year period, and validation and testing data each cover an entire year. This way, all trends and seasonal variations are captured. According to Pasero, Raimondo & Ruffa (2010), long-term out-of-sample prediction is a more challenging task, as the model has to deal with growing uncertainty as it needs to make predictions farther into the future. We see this in the overview and our results as well: using out-of-sample data leads to lower predictive accuracy. The second possible reason that we identify is the limited amount of data used in some of these publications; e.g. Kamruzzaman et al. (2003) achieve significantly better results than we do, but use only 565 observations. We believe this opens up the possibility that perhaps the test data was unusually similar to the training data, inflating test accuracy; however, this is only an assumption. In order to rule out this possible influence, we use a large amount of data, over 2.200.000 observations. This is detailed further in the next section.

A final remark concerns the results of Di Persio et al. (2016). While it is unclear whether they use in- or out-of-sample data, they do mention normalising their data before splitting in training, validation and test set; which allows the algorithm to use information from validation and test set, and should not be done. It will also inflate test accuracy.

No	Publication	Type of time series	Inputs	Granularity	Number of observations	Out-of-sample	Methods applied	Performance (Accuracy)	Performance (other)
1	Chao, Shen & Zhao (2011)	FOREX	Close	Weekly	937	Yes	MLP	41,74%	RMSE:9,41E-03
							DBN	**57,55%**	RMSE:8,21E-03
2	Yao & Lim Tan (2000)	FOREX	Close and Technical indicators	weekly	510	Yes	ARIMA	55,86%	
							MLP	**56,00%**	**NSME:0,11**
3	Kamruzzaman & Sarker (2003)	FOREX	Close	Weekly	565	Yes	ARIMA	52,94%	NMSE:1,0322
							MLP	**83,08%**	**NSME:0,0441**
4	Kayal (2010)	FOREX	OHLC and Technical indicators	Hourly	595	**No**	Random		32,08% winning signals
							MLP		45,45% winning signals
5	Vaiz & Ramaswami (2017)	Stock	OHLC and Technical indicators	Daily	+/- 1000	**No**	MLP	88,33%	
							SVM-MLP (hybrid)	**90,00%**	
6	Martinez, da Hora, Palotti, Meira & Pappa (2009)	Stock	OHLC and Technical indicators	Daily	1283	**No**	MLP		MAPE:2,79%
7	Gunduz, Yaslan & Cataltepe (2017)	Stock	OHLC and Technical indicators	Hourly	6703	Yes	LR		F-measure:0,545
							CNN		F-measure:0,544
							CNN with grouped features		**F-measure:0,563**
8	Armano, Marchesi & Murru (2005)	Stock	Tis	Daily	2000	yes	GA-MLP hybrid	55,50%	
9	Galeschuk & Mukherjee (2017)	FOREX	Close and Technical indicators	daily	1565	yes	ARIMA	60,05%	
							MLP	66,23%	
							CNN	**79,27%**	
10	Vargas, de Lima & Evsukoff (2017)	Stock	Text and Technical indicators	daily	17174	yes	SI-RCNN	62,03%	
11	Di Persio & Honchar (2016)	Stock index	Return price	Daily	16706	**Unclear**	MLP	60,00%	
							CNN	59%	
							RNN	61%	
							Wavelet-CNN	**62%**	
		FOREX	Return Price	Per minute	372210	**Unclear**	MLP	81%	
							CNN	81%	
							RNN	79%	
							Wavelet-CNN	**83%**	
12	Chen, Zhou & Dai (2015)	Stock	Close and Technical indicators	daily	1,211,361	yes	LSTM	27,2% (7 categories)	
13	Patel, Shah, Thakkar & Kotecha (2015)	Stock	Technical indicators	daily	2474	**no**	ANN	86,69%	F-measure:87,14%
							SVM	89,33%	F-measure:89,52%
							RF	89,98%	F-measure:90,14%
							Naive-Bayes	**90,19%**	**F-measure:90,50%**
14	Nelso, Pereira, de Oliveira (2017)	Stock	OHLC and Technical indicators	per 15 minutes	last +/- 6400	**No**	LSTM	55,90%	
15	Guresen, Kayakutlu & Daim (2011)	Stock index	Close	Daily	182	yes	MLP		MSE:2478,15
							GARCH-MLP		MSE:3665,84
							DAN2		**MSE:1472,28**
							GARCH-DAN2		MSE:20901,2

3. Input data

3.1 Technical and fundamental analysis

Several different types of data can be used as inputs for financial time series prediction. Patel, Shah, Thakkar & Kotecha (2014) define two different kinds of input data. On the one hand, there is fundamental data. Fundamentalists, or investors using this data, look at the intrinsic value of the financial product, as well as the political and economic climate. Based on this they will decide the value of a product, and compare it to its current price. If the price is lower than the estimated value, it is considered to be an interesting investment. Technical data, on the other hand, consists of statistics that are generated based on market activity, such as the price and sold volume of stock and rates. Rather than trying to determine the value of a product, an assessment is made of the likelihood that its price will go up or down. Technical analysis can further be split in two streams. First there is the graphical analysis, where an investor will consider several graphs concerning a certain stock or rate, and make a guess on whether the price will rise or fall based on this. According to Martinez, da Hora, Palotti, Meira & Pappa (2009) this is highly subjective, and it does not allow to easily compare and use several indicators. In the second stream, several technical indicators are calculated and used as input data in an algorithm. Vanstone & Finnie (2009) argue that when making short-term predictions, technical analysis should be favoured over fundamental analysis.

3.2 Technical indicators

As we will be trying to classify whether price will go up or down for our exchange rates on a per-minute basis, we will take the aforementioned recommendations into account, and use several technical indicators as inputs for our different deep learning algorithms. Because we do not claim to have any expert knowledge on which technical indicators will have the best predictive power, it was decided to consult the literature in order to get a picture of what are common technical indicators. Table 3.1 below sums up the indicators used, what time-span they are based on, the type of phenomenon the indicator is indicative of, as well as the formula for the indicator. Some formulas are calculated in several steps, the helper functions for these steps are listed in table 3.2.

Table 3. 1: Technical indicators used as inputs

Technical indicator	Time period used (minutes)	Indicator type	Formula
Simple moving average	5, 10, 20	Smoothing	$SMA = \sum_{i=0}^{n-i} \frac{Close(i)}{n}$
Exponential moving average	5, 10, 20	Smoothing	$EMA = [Close(i) - SMA(i)] * \frac{2}{(n+1)} + SMA(i-1)$
Double exponential moving average	5, 10, 20	Smoothing	$DEMA = 2 * EMA - EMA(EMA)$
Weighted moving average	5, 10, 20	Smoothing	$WMA = \frac{Close(i) * n + close(i-1) * (n-1) + \cdots + close(i-n) * 1}{n * (n-1)/2}$
Moving average convergence/divergence	fast=12; slow=26	Momentum	$MACD = EMA(12) - EMA(26)$
Williams R%	14	Oscillation	$\%R = \frac{(Highest\ high(14) - Close(i))}{(Highest\ high(14) - Lowest\ low(14))} * (-100)$
Stochastic oscillator %K	14	Oscillation	$\%K = \frac{(Close(i) - Lowest\ low(14))}{(Highest\ high(14) - Lowest\ low(14)} * 100$
Commodity channel index	20	Trend	$CCI = \frac{TP(i) - SMA(TP)}{0.015 * MD}$
Relative strength index	14	Oscillation	$RSI = 100 - \frac{100}{1 + RS}$
Standard deviation	5, 10, 20	Variation	$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$
Upper Bollinger band	20	Variation	$UB = SMA + 2 * \sigma(20)$
Lower Bollinger band	20	Variation	$LB = SMA - 2 * \sigma(20)$
Average true range	14	Volatility	$ATR = \frac{\sum_{i=0}^{n-i} TR(i)}{14}$
Directional movement index	14	Momentum	$DX = 100 * \frac{\lvert +DI - -DI \rvert}{+DI + -DI}$
Average directional index	14	Momentum	$ADX = 100 * \frac{WSM(\lvert +DI - -DI \rvert)}{+DI + -DI}$
Aroon indicator up	25	Trend	$\text{Aroon} - \text{up} = 100 * \frac{25 - days\ since\ 25day\ high}{25}$
Aroon indicator down	25	Trend	$\text{Aroon} - \text{down} = 100 * \frac{25 - days\ since\ 25day\ high}{25}$

Chande momentum oscillator	14	Momentum	$CMO = 100 * \frac{\sum(gain - loss)}{\sum(gain + loss)}$
Momentum (Rate of change)	10	Momentum	$ROC = 100 * \frac{Close(i) - Close(i-10)}{Close(i-10)}$
Rate of percentage change	10	Momentum	$ROCP = \frac{Close(i) - Close(i-10)}{Close(i-10)}$
Percentage price oscillator	fast=12; slow=26	Momentum oscillator	$PPO = 100 * \frac{EMA(12) - EMA(26)}{EMA(26)}$
Ultimate oscillator	short=7; medium=14; long=28	Momentum oscillator	$UO = 100 * \frac{4 * average7 + 2 * average14 + average28}{7}$

Table 3. 2: Helper functions for the technical indicators

Helper function	Time period used (minutes)	Formula
Typical price	1	$TP = \frac{High(i) + Low(i) + Close(i)}{3}$
Mean deviation	20	$MD = \frac{\sum_{i=0}^{n-i} \lvert TP(i) - SMA(TP) \rvert}{20}$
Average gain/loss	14	$avg\ gain/loss = \frac{\sum_{i=0}^{n-i} gain/loss(i)}{14}$
Relative strength	1	$RS = \frac{average\ gain}{average\ loss}$
True range	1	$TR = High(i) - Close(i)$
Positive directional movement	1	$+DM = \max(High(i) - High(i-1), 0)$
Negative directional movement	1	$-DM = \max(Low(i) - Low(i-,), 0)$
Positive directional indicator	14	$+DI = 100 * \frac{WSM(+DM)}{WSM(ATR)}$
Negative directional indicator	14	$-DI = 100 * \frac{WSM(-DM)}{WSM(ATR)}$
Buying pressure	1	$BP = Close(i) - Close(i-1)$
Average7	7	$\frac{\sum BP}{\sum TR}$
Average14	14	$\frac{\sum BP}{\sum TR}$
Average28	28	$\frac{\sum BP}{\sum TR}$
WSM		Wilder's smoothing techniques

This selection of indicators is based on which indicators seem to appear most in the consulted literature on time-series forecasting with deep learning, which was covered in the previous section. We do not pretend that this is the best selection of technical indicators, but this does not seem to be necessary to get good results. When comparing neural networks with statistical techniques, Martinez et al. (2009) state that "the main advantage of these techniques over the statistical ones is their ability to explore the tolerance of systems to data uncertainty, imprecision and partial truth." In a similar way, Kayal (2010) argues that including some additional technical indicators or omitting some of the ones that were included, is unlikely to heavily affect the prediction results. Aside from these technical indicators, we also included the historical open, close, high and low prices of the rate. The sold volume per minute was not considered, as the FOREX market is decentralized, and thus no precise estimations can be made based on the volume (Kayal, 2010). Consequently, also no technical indicators based on the volume were used. All data was downloaded from the website of Dukascopy, a Swiss FOREX trading bank, through their historical data feed tool (Dukascopy, historical data feed, 2018). When downloading the data, we filtered out the flat periods, in which no trades were made. Usually trades are made on the FOREX market on a 24 hour per day basis, from 22:00 GMT on Sunday, when the market opens in Sydney, until 22:00 GMT on Friday, when markets close in the US. The formulas of the technical indicators were adapted from stockcharts.com (Stockcharts, technical indicators and overlays, 2018).

3.3 Input data and data preparation

The combination of these technical indicators, for each of the different assessed time periods and the historical data (open, close, high and low prices), provide 36 inputs to feed into the different algorithms. As we try to predict whether the exchange rate will go up or down, a dependent variable was created, for each observation, taking a value of 1 when the closing price of the rate was higher the next minute, and a 0 when it was lower. This essentially turns our problem into a binary classification problem. To make a prediction for a certain minute, not only the current minute's data was used, but also the 36 inputs of the past 39 minutes. This results in a time window of 40 minutes' worth of 36 inputs to predict a 1 or 0 for the next minute. Each observation thus consists of one 40x36 window of independent data. How the different algorithms handle these windows is discussed in their respective next sections.

Before feeding the input data into the models it is important to perform one final step: input scaling. Scaling is done for a couple of reasons. First, it helps to avoid numerical problems: when certain values are very large, then the weights in the network might remain fairly small, and updates will not have a large effect, while simultaneously possibly reducing training time (Ballings et al., 2015). Second, it eliminates possible side effects resulting from range differences in values between different inputs (Akita, Yoshihara, Matsubara & Uehara, 2016). Gunduz et al. (2017) stress the importance of not learning from the test data by calculating the scaling parameters based on the training data. While other methods are possible, we scaled our data by standardisation, by calculating the mean μ and variance σ^2 of the input training data, and applying standardisation formula to all datapoints x_i.

$$Z_i = \frac{x_i - \mu}{\sigma^2}$$

For all of our currency pairs, we used 4 years of per-minute data for training, 1 year for validation and 1 year for testing. The validation set is used for early stopping, and the test set is used for assessing final performance of the model. For all of our currency pairs the data from 01/01/2012 until 31/12/2015 was used for training, and the data from 01/01/2016 until 31/12/2016 and 01/01/2017 until 31/12/2017 were used for validation and testing, respectively. This equals about 373,300 data points per year. The training set contains about 1,493,000 observations.

All of the deep learning models in this thesis were finetuned and assessed for EUR/USD currency pair. In order to be able to evaluate how well the models perform on other currency pairs we trained the architectures for other currency pairs as well. The other chosen pairs are: GPD/USD and JPY/USD. The results of these predictions will be discussed in section 5.3. The exact amount of data for all currency pairs, per set is outlined below in table 3.3.

Table 3. 3: number of observations per data set, per currency pair

Currency pair	Training set	Validation set	Test set	Total
EUR/USD	1,493,140	373,295	373,296	2,239,771
GBP/USD	1,493,140	373,295	375,384	2,241,859
USD/JPY	1,493,140	373,295	373,862	2,240,337

Due to differences in Holidays for the countries to which currencies belong, etc, the actual sizes of the different datasets differ a bit. We opted to use the same size for the training and validation set, for all currency pairs. This means that at some points it is possible that a certain set is slightly

larger than it is supposed to be; e.g. the test set for the GBP/USD currency pair likely contains some data on the last couple of hours of December 30th, 2016. While this could have been solved, we believe the effects of this will be minimal to non-existent, due to the large amount of data. Note that the training set should have contained 40 more observations; these were dropped to wash out the effect of initialising the values of the technical indicators.

4. Neural networks

Before discussing the prediction results, we will explore the architectures and characteristics of the different neural networks used in this thesis, starting with a general description of artificial neural networks. In this section, we will introduce the different networks, as well as go deeper into some concepts that will prove to have a large influence on the prediction results.

4.1 Artificial neural networks

4.1.1 Basic architecture and training

The basic building block of an artificial neural network is the neuron (or node). These neurons work as information processors, and were originally inspired by human brain cells (Kayal, 2010). The neurons can be placed in different layers, and are connected to each other by weights, making up a neural network. An example of a feedforward neural network, or multilayer perceptron is shown below.

Figure 4. 1: A feedforward neural network

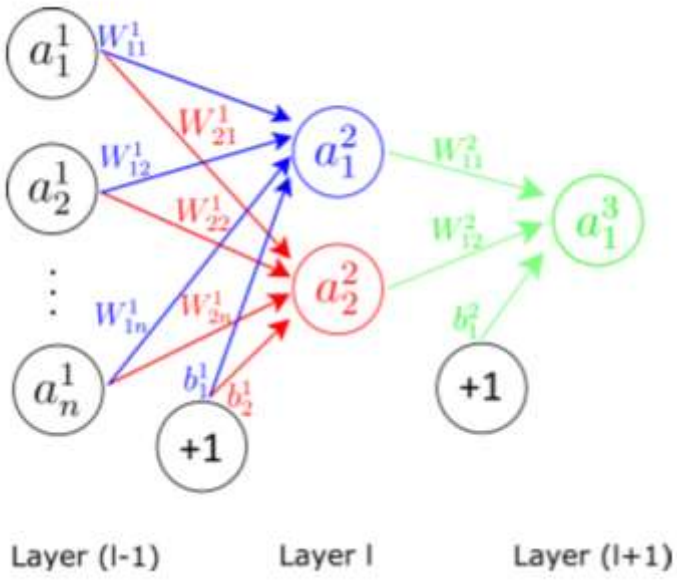


Image adapted from Ordóñez et al. (2016)

In this figure we have an MLP with 1 input layer, consisting of n inputs, one hidden layer (the middle layer), and one output layer, consisting of 1 output neuron. Networks consisting of 2 or more hidden layers are called deep neural networks. Typical for MLPs is that all nodes are connected to all nodes in the previous and following layer, which is why they are often called dense or fully-connected networks (Ordóñez et al., 2016).

These networks are trained in a supervised manner, consisting of two steps or passes. In the forward pass, the value of each node is calculated as follows: first the weighted input is calculated, by multiplying the value of the previous node with the weight, and summing for all the incoming connections, and then applying a non-linear function to this sum (LeCun et al., 2015). These non-linear functions were traditionally sigmoids or hyperbolic tangents (tanh), although more recently the rectified linear units (ReLU) have become very popular. These non-linear functions, often called activation functions, will prove to be important to the predictive performance of the MLP networks. Predictive performance of MLP networks is discussed in section 4.3, and the influence of activations functions on their performance is further explored in section 5.2. The activation function of the output nodes is often different from the activations elsewhere in the network. Because our problem is a classification problem, we use a softmax activation function for the output nodes. This will result in the network calculating probabilities that the closing price for the next minute will be higher or lower than this minute's, and those probabilities will add up to 1 (Géron, 2017).

In the second step, the backward pass, the error at the output is calculated by measuring the difference between the predicted and the desired output. Then the algorithm determines to what extent the neurons in the previous layer contribute to the error, and this is done all the way through the network; hence the name of the training algorithm, backpropagation, which was first proposed by Rumelhart, Hinton & Williams (1986). The error is said to be backpropagated through the network, as it is calculated at the end of the network, and then, while adjusting the weights, propagated all the way to the input layer. The error is not calculated as an exact value, but rather as a first-order derivative of the output of each unit. At the output unit, where the algorithm starts, this is simply done by differentiating the cost function, after which the derivative for each unit in each earlier layer can be determined by means of the chain rule for derivatives (LeCun et al., 2015). When the derivatives have been calculated, the corresponding weights are adjusted according to the steepest gradient descent (Géron, 2017). The derivative is the gradient of the local error, and the weights are adjusted in the opposite direction of the gradient. The extent to which the weights are adapted is determined by the learning rate. According to Zeiler (2012), there are two main pitfalls concerning the learning rate; the first is that one might choose a learning rate that is too large, leading the algorithm to overshoot the global minimum, possibly never converging to the

best solution. The second is choosing a learning rate that is too small resulting in a slow learning process. Gradient descent and the influence of the learning rate are illustrated in the image below.

Figure 4. 2: Gradient descent and the learning rate

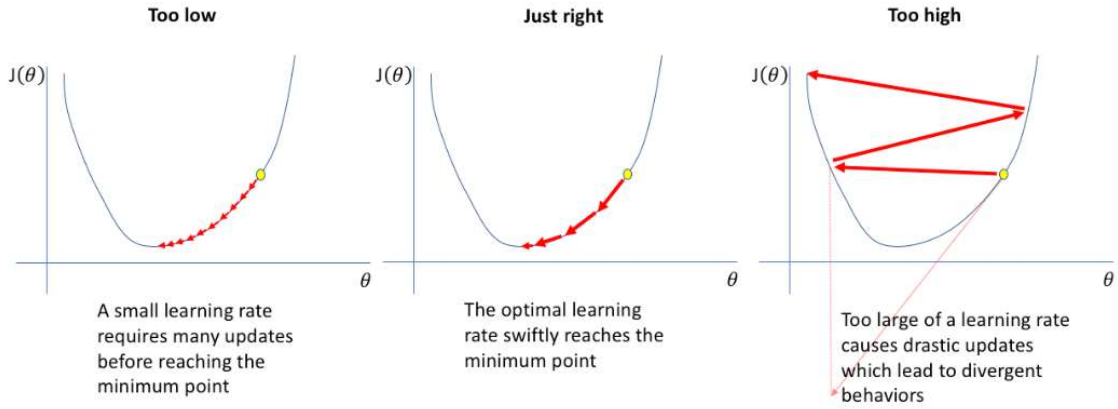


Image adapted from (Jordan, 2018)

These graphs are a simplification of what happens after the gradient is calculated. The weights are adjusted in the opposite direction of the steepest descent, moving closer to the global minimum. It also illustrates the pitfalls: setting a learning rate too low will unnecessarily prolong training, a learning rate to high might never converge to a solution.

Another aspect that might potentially slow down training is using the entire training set before adjusting the weights. Most often, instead of using the entire training set, the errors will be calculated after a certain part of it has been processed (batch learning), or after randomly selecting several training observations (stochastic gradient descent) (Zeiler, 2012). In doing so, the learning time can be significantly reduced. One round of putting all training observations through the model, either in full or in batches, is called an epoch, or an iteration. Due to the size of our training set, and even test and validation sets, we had to use batches during training, since especially in training CNNs using the entire set would lead to out-of-memory errors. The ideal size of these batches will be discussed in the section on LSTM networks.

4.1.2 Training optimization algorithms

Several optimization algorithms have been developed with the goal of reducing training time and increasing accuracy. These algorithms often use an adaptive learning rate, changing the learning rate depending on how close the algorithm is to the global optimum, taking bigger steps early on,

and decreasing the learning rate as the error becomes smaller. In this thesis we consider 3 possible optimization methods. The first method considered is the Momentum Optimizer, which is a simple adaptation of stochastic gradient descent, proposed in Rumelhart et al. (1986). It uses an exponential decay parameter, which keeps track of past parameter updates. When the gradient consistently points in the same direction, early on in training, larger steps will be taken. When the gradient direction starts to vary more, the learning rate is decreased, to get a more fine-tuned adjustment of the weights (Zeiler, 2012). During training of the LSTM neural networks, we found however that this optimizer behaved in an unstable way at times, leading to 0 validation accuracy. Consequently, we decided not to use it for the other networks.

Another tested optimization algorithm was the RMSProp optimizer, as proposed in the famous coursera lecture by Tieleman & Hinton (2012). They propose to adjust the learning rate for every weight by dividing the learning rate by a moving average of the squared gradient.

The third optimizer used was Adam, which stands for adaptive moment estimation, and was proposed by Kingma & Ba (2014). It is inspired by both RMSProp and AdaGrad, another successful optimizer, by keeping track of both the decaying past squared gradients of a weight, as in RMSProp, as well as the decaying average of past gradients, as in AdaGrad (Géron, 2017). According to Kingma et al. (2014), Adam is the ideal optimizer for large datasets, working well for both MLP networks and convolutional neural networks. As we will discuss in further detail in section 4.2, we performed a randomised grid search to find the best long short-term memory network for our problem, which showed that Adam was the best optimizer. Preliminary tests, however, lead us to believe that there is little to no difference in performance when either RMSProp or Adam is used. Because of the advantage cited by Kingma et al. (2014) and the result of the grid search, the Adam optimizer was used for all other networks as well.

4.1.3 Regularization

In their paper on the bias and variance trade-off for neural networks, German, Bienenstock & Doursat (1992) argue that complex neural networks will always be prone to high variance, with the theoretically ideal case of an infinite amount of training data being the only exception. When a model has high variance rather than high bias, it means that it is learning the noise in the data, rather than the actual relation between the input features and the classes it is supposed to predict.

Consequently, when the model is applied to new inputs, which were not used in training, it will fail to make good predictions. Even though we use a large amount of training data in our tests, the networks are still very complex, and appear prone to overfitting. As we will discuss later, most of the CNNs could be trained in 1 epoch or less, and the LSTM and MLP networks were easily trained within 10 epochs. In order to prevent the network from overfitting, we applied two methods that we encountered often in the literature on deep learning. The first method, dropout, was used by Honchar et al. (2016) and Gunduz et al. (2017). The second method, early stopping, was applied by Yuan et al. (2016), Gunduz et al. (2017) and Lipton et al. (2017).

4.1.3.1 Dropout

Dropout is a very simple yet effective way to prevent overfitting in neural networks. The method was first proposed by Srivastava, Hinton, Krizhevsky, Sutskever & Salakhutdinov (2014), and essentially consists of "dropping" the connections between neurons in consecutive layers during training. Every time a new batch is put through the model in the training phase, every neuron in the model is dropped with a certain probability. A dropped neuron will not be used while training that batch, and is effectively temporarily removed from the network, as shown in figure 4.3 below.

Figure 4. 3: Dropout

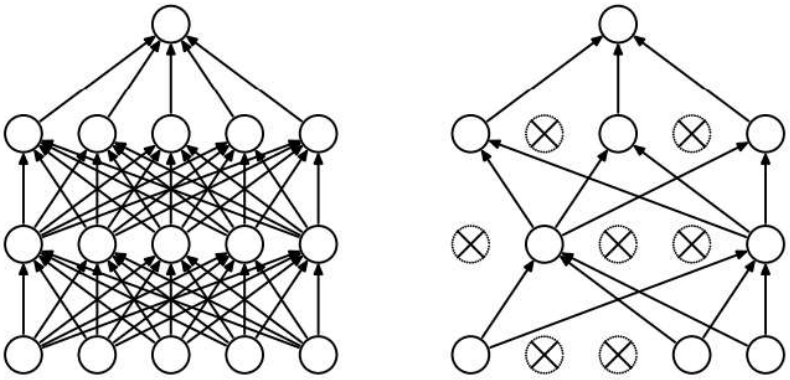


Figure adapted from Srivastava et al. (2014)

Hinton et al. (2012) argue that the method works as it prevents the neurons in the network from co-adapting, i.e. work well together to learn the training data to such an extent that the model overfits. By applying dropout, each neuron learns a certain feature in the data, and contributes independently to providing the right answer. Srivastava et al. (2014), further explore the method,

finding that not only does it reduce overfitting, it also allows for faster network training. In their benchmark tests, they find that it works better than other regularization methods. According to LeCun et al. (2015), dropout contributed to recent successes in lowering generalization errors in image classification accuracy. Even though the dropout probability is a parameter to be finetuned while training, Srivastava et al. (2014) recommend starting with 50%.

4.1.2.2 Early stopping

Early stopping is another common regularization technique, and is interesting for its simplicity and intuitive nature (Prechelt, 1998). In this technique training is done on the training set, and at regular intervals the error is calculated on the validation set. Every time the error on the validation set reduces, the network weights are saved. When the error on the validation sets keeps on increasing again, and has increased for a predetermined amount of epochs, training is stopped, and the model with best validation is restored and used to calculate final test accuracy.

Prechelt (1998) argues that there is no general rule that dictates when early stopping should be applied, and choosing when to stop early is mostly a trade-off between generalization error (how well does the model perform on unseen data, i.e. on the validation set), and training time. In this thesis, we mostly allowed a larger amount iterations before early stopping for the LSTM and MLP networks, as these networks tended to have larger fluctuations in validation error, and learned relatively fast, making it worthwhile to learn a little longer. For the CNN and CLSTM networks, the validation error seemed to increase again fairly shortly after training started, while training accuracy kept on going up, indicating that these models tend to overfit, and they even did with high dropout probabilities. Due to this, in combination with their longer training times, we usually stopped training fairly early. This approach is very ad hoc, however, based mostly on our own experience.

4.2 Long short-term memory network

Already having established the appeal of LSTM networks, as well as their advantage over other recurrent neural networks (cfr. Section 1.3.1), we will now delve deeper into the typical LSTM architecture, and then discuss some of the LSTM models built for this thesis, as well as present a randomized grid search we performed to find a good architecture.

4.2.1 Long short-term memory network architecture

Recurrent neural networks work very similar to the traditional feedforward neural network, discussed earlier in section 4.1. The main difference lies in that it also has recurrent connections, as can be seen on the figure below.

Figure 4. 4: A recurrent neural network

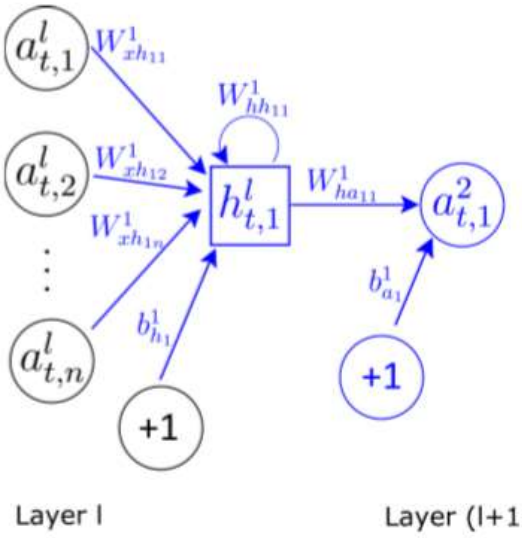


Image adapted from Ordóñez et al. (2016)

These recurrent connections allow the network to not only use input data, but also retain some information learned from previous input data (Graves et al., 2013). The recurrent connection weights are trained similarly to all other weights, determining how much information the network uses from previous inputs. The vanishing gradient problem mentioned earlier often renders these models incapable of learning to bridge long time lags, as it will either not work at all, or take a prohibitively long time (Hochreiter et al., 1997). The LSTM architecture proposed by Gers et al. (1999) is now considered to be the "vanilla" LSTM network (Yuan et al., 2016), and is the architecture used in this thesis. The neurons in the network contain input and output gates as presented in the original model, and are improved on as they include the so-called forget gates. The architecture of such a neuron can be seen in figure 4.5 below. The network is built up of these neurons, similar to an FFNN.

Figure 4. 5: A LSTM cell

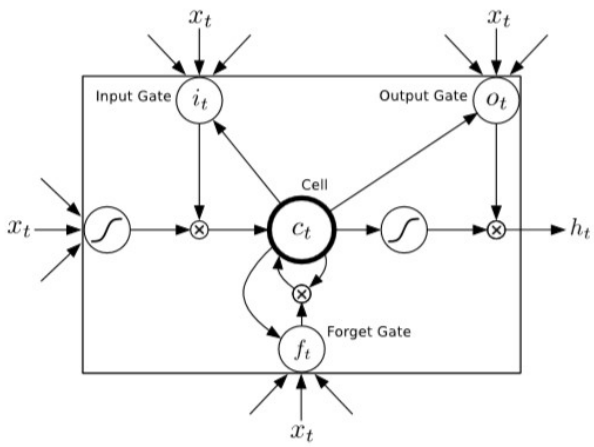


Image adapted from Graves et al. (2016)

An LSTM network operates in a similar way as the MLP network presented earlier: the LSTM-cell shown in figure 4.5 works as a neuron in the earlier shown images. Every time an observation is passed through the model, a cell receives the outputs of the previous layers as its inputs. These inputs are first put through an activation function, and then the input gate acts as a filter, determining how much the internal state is allowed to be perturbed by the input (Hochreiter et al., 1997). As it is a recurrent network, the inputs to the model are both the new input, as well as the previous inputs about which information was retained by the network. Then, in the architecture proposed by Gers et al. (1999), the forget gate determines to what extent the cell memorises or forgets these inputs, adapting the cell state. This state is again put through a non-linear function, and finally the output gate determines to what extent this cell will pass info on to the next layer. The main advantage of LSTM networks over other RNNs lies in this internal cell state, which is essentially a recurrent self-connected unit. In the original LSTM architecture this unit had a weight of 1, which solved the vanishing gradient problem, as this prevents the error from blowing up or vanishing, even when the unit is presented with no new input. This mechanism is called the constant error carrousel. The input and output gate protect the internal cell state from the forward and backward flowing error respectively. The difference between the original architecture and the architecture proposed by Gers et al. (1999) lies in the fact that this weight is no longer always 1, but instead a weight that is trained like all others. This allows the network to learn continuous time series that are not segmented (Gers et al., 1999).

4.2.2 Preliminary LSTM network tests

LSTM networks were the first neural networks we applied to our FOREX direction prediction problem. For the very first network we made, we looked at the literature, to get an idea of what are common hyperparameter values. Hyperparameters are parameters that determine the topology of the network and how it learns. Examples include the number of layers in the network, the number of neurons or LSTM-cells per layer, the learning rate, activation functions and the optimizer. All of the tests to determine the best network architecture were done on the EUR/USD currency pair, in this section, as well as for all other models. Section 5.3 is dedicated to applying the optimized networks to other currency pairs and seeing how well these architectures generalise to other pairs.

As discussed earlier, we will use a time-window approach for the inputs. The LSTM network is presented with data from a certain minute, as well as the data of the past 39 minutes, to make a prediction whether the exchange rate will rise or fall. For the very first model we used only 2 hidden layers, as this is found to be enough in some cases (ElSaid, Wild, Higgins & Desell, 2016), and each layer contains 20 neurons, which was the amount of inputs we had at the time; we increased this to 36 during preliminary testing, in an attempt to increase the accuracy. While some authors use smaller minibatches, such as 30 in Honchar et al. (2016) or 64 in Chen et al. (2015), we choose a higher number, 1000, due to the fact that we have about 1.493.200 observations in the training set, and believe using smaller batches might prolong training time. We did not use early stopping or dropout for this first model, which resulted in a test accuracy of 52.35% after training for 25 epochs, with the Adam optimizer.

After this first model, a fair amount of different combinations of hyperparameters were tested. Early stopping was used after 10 iterations of rising error on the validation set, reverting back to the best parameters to calculate test accuracy. Several different methods were explored in an attempt to increase the accuracy on the test set; we successfully experimented with larger batch sizes, as Smith, Kindermans, Ying & Le (2018) found that using larger batch sizes reduces training time, as well as potentially increases accuracy. We found this to be true in our case as well; increasing the batch size from 1000 to 10.000 lead to an increase in accuracy of 0.1 percentage point in one case. As recommended by Srivastava et al. (2014), a dropout rate of 50% was used as a starting point, also leading to higher accuracy.

The effect of increasing the number of layers or the amount of units per layer was less clear-cut: networks with more layers and more units tended to overfit more, and required quite some finetuning in dropout rate. Models with 4 layers, but higher dropout seem to perform equally well compared to models with 2 layers and lower dropout, ceteris paribus. Ex post we also looked into the use of different activation functions, but had to conclude that the effect on the predictive performance is minimal, which we will explore in a next section 5.1, where we compare LSTM and MLP network performance. Initially, however, we settled on using the ReLU activation function, as Krizhevsky, Sutskever & Hinton (2012) argue that when using backpropagation non-saturating non-linearities will lead to faster training compared to saturating non-linearities, such as the more traditional sigmoid or tanh functions. All together 30 models were tested, which gave some intuition about different possible values for the hyperparameters. The test accuracies of these models ranged from 50.58% to 52.70%.

4.2.3 Randomized grid search

As there is a virtually infinite amount of possible combinations of different hyperparameters, we decided to perform a grid search to find the best set of hyperparameters. Based on our initial tests we established different values to be tested for each parameter, outlined below, in table 4.1.

Table 4. 1: Hyperparameter values for LSTM network grid search

Hyperparameter	Tested values
Number of values	36; 72; 108
Number of layers	2; 3; 4
Batch size	5,000; 10,000; 20,000
Learning rate	0,01; 0,001; 0,0001
Optimizer	Adam; RMSProp; Momentum
Dropout rate	0,3; 0,5; 0,7

To decrease the number of models to be tested, it was decided to use the same number of neurons in every layer. An exhaustive grid search of all these possible values would lead to a total number of 729[1] models to be tested. In order to find a model that works well on unseen data, k-fold cross validation can be used (Kohavi, 1995). In k-fold cross validation, the data is split randomly into k folds, and trained k times on k-1 folds of the data, and validated on the hold-out fold. Each

[1] 6 hyperparameters, with 3 differing values each, leads to $3^6 = 729$ models.

fold is held out once. The model with the highest accuracy on the validation set is considered best, and the test accuracy is reported as definitive model performance. The downside to this approach is that it multiplies the number of models to be tested by k. As we were limited in time and resources, we opted for a randomized grid search, where a predetermined number of models are tested, selecting random values for each parameter, rather than exhaustively going through the entire search space. Bergstra & Bengio (2012) argue that in most cases this approach leads to models that are at least of comparable quality to models found in an exhaustive search, while using only a fraction of the computation time. While using 10-fold cross-validation seems to be the rule of thumb, other values for k are used as well, such as Gunduz et al. (2017), using only 5 folds. Because a trade-off needs to be made between how thorough a model is checked, and how many models can be checked, we decided to perform 3-fold cross validation, for 40 randomly selected models, totalling 120 tested models. Another measure taken to limit the training time was to limit each model to only 10 training epochs, and early stopping after the validation error rose for 2 epochs. We used the Scikit-learn library for the search (Pedregosa et al., 2011). More specifically, the RandomizedSearchCV function was used to randomly select the hyperparameters, run all the models and keep track of the best model. Appendix A gives an overview on how the code is structured, and appendix K contains the code for the randomized grid search.

Training all of these models took 13050 minutes (slightly over 9 days) on a system with 40 cores, 12 of which were dedicated to the program, and 400 GB of RAM. The randomized grid search found that out of all tested models, a model with 2 hidden layers each containing 36 neurons, with a learning rate of 0.01, batch size 5000, dropout rate of 50% and the Adam optimizer performed best, resulting in a test accuracy of 52.71% on the EUR/USD currency pair. In order to assess the robustness of the model, we trained it again, this time for a maximum of 100 epochs, with early stopping after 10 epochs, and this 3 times. The results are summarised in table 4.2 below.

Table 4. 2: Repeated training of optimized LSTM model

Run	Test accuracy	Training time (seconds)	Epochs trained
1	52,70%	9024	16
2	52,70%	10920	21
3	52,70%	11222	19
Average	52,70%	10389	18,67

The main conclusion from this test is that this model is indeed very robust, as put forward by LeCun et al. (2015).

The code of the randomized grid search is attached in appendix K; the sample code of simply running one LSTM model is attached in appendix H.

4.3 Multilayer perceptron

Since publications that explore either CNN or LSTM networks for financial time series forecasting often include MLP networks, we will do so too, to offer a more complete comparison. Examples of such publications are Honchar et al. (2016) and Galeschuk et al. (2017). For this comparison we decided to simply use the network architecture that proved most successful for LSTM in our problem, but replace the LSTM neurons with regular ones, forming a dense neural network. The proposed architecture thus features 2 hidden layers, each containing 36 neurons, with a 50% dropout chance, early stopping after 10 iterations without improvement, learning with a learning rate of 0.01, batch size 5.000, ReLU activations and optimized with the Adam algorithm. It should be noted that opposed to LSTM networks, MLP networks can not capture time dependencies. The inputs are not fed into the network as a 40x36 window, as is done for all other networks, but rather as a 1x1.440 input. The input layer has one node for each of these values. The order of the input nodes does not affect predictive performance, since all input nodes are connected to all nodes in the following layer. The results of running this model three times can be found in the table below.

Table 4. 3: MLP test performance with ReLU activation function

Run	Test accuracy	Training time (seconds)	Epochs trained
1	52,70%	472	11
2	52,70%	394	11
3	52,70%	371	10
Average	52,70%	412	10,67

It would appear that contrary to our hypothesis that all other networks would outperform MLP, our LSTM network does not outperform an MLP network with similar architecture; performance is almost exactly the same, and this is achieved in much less training time: averaging 412 seconds for MLP compared to 10.389 seconds for LSTM networks, or about 4%. A possible explanation for the high performance of MLP networks is put forward by Glorot, Bordes & Bengio (2011). They find that using ReLU activations rather than the more traditional sigmoid or tanh activation

functions leads to a significant increase in accuracy, when using MLP networks for image recognition. As our problem is a classification problem as well, we decided to do an ex post test of the influence of the activation function on test accuracy. We detail the results of these tests in section 5.2 and find that indeed, while LSTM networks seem largely unaffected by the choice of activation functions, the MLP network performs significantly worse when using either sigmoid or tanh activation. We see this in our literature on deep learning for financial time series forecasting as well: Galeshchuck et al. (2017) report using a sigmoid activation for their MLP, which underperforms compared to the CNN network (see table 2.1). Di Persio et al. (2016), on the other hand, use ReLU activations for both their CNN and MLP models, and sigmoid activations for their LSTM networks. When forecasting direction of the S&P500 index, their LSTM narrowly outperforms the MLP network, and when forecasting FOREX, their MLP model even outperforms their LSTM model. While the publications mentioned here do not intentionally explore different activation functions, and how they influence comparative performance between MLP and LSTM, they are in line with our finding: The choice of the activation function heavily influences MLP performance for financial time series forecasting.

4.4 Convolutional neural network

Similar to the discussion on long short-term memory networks, we will first discuss the convolutional neural network architecture, before proceeding to our tests and the results.

4.4.1 Convolutional neural network architecture

As mentioned earlier, convolutional neural networks are traditionally made up of two parts: the first part consists of multiple consecutive convolution layers, some of them separated by maxpooling (subsampling) layers. The second part is then usually an MLP, that takes the output of the last layers as its inputs, and its last layer will make a prediction. Figure 4.6 below is a schematic representation of the LeNet-5 architecture, presented by LeCun et al. (1998).

Figure 4. 6: A convolutional neural network

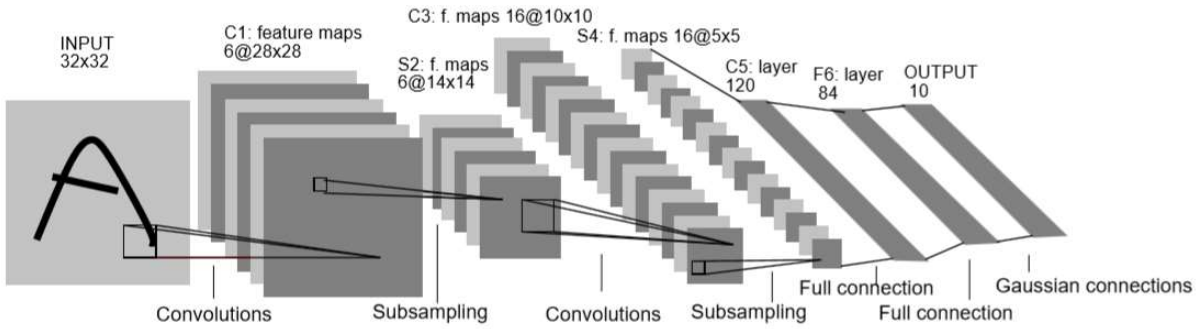


Image adapted from LeCun et al. (1998)

In the image we see the input, which is a 32x32 image. It is scanned by the nodes in the feature maps in the subsequent convolutional layer. In this case, this layer contains 6 feature maps. Each of these maps has a size of 28x28. These maps are essentially planes of neurons that share a single weight vector (Lecun et al., 1998). Each neuron receives inputs from within its receptive field, a small neighbourhood (e.g. 5x5) of neurons in the preceding layer. The size of the receptive field is also called the kernel size. In the case of 6 feature maps, this means that the 6 neurons with the same location on the different layers will extract 6 different features from the previous layer. As the data flows through the different layers, these features are enhanced more and more. Hence, the convolutional neural network is said to have its own feature extractor (Lecun et al., 1998). In number image recognition, for example, such features can be edges or end-points in numbers.

Since the exact location of a feature is less important than its relative position to other features, the robustness of the network can be increased by adding subsampling (maxpooling) layers (LeCun et al., 1998). These layers will take the average or maximum value of the values of the neurons in the preceding feature maps, and multiply them with a trainable coefficient. This operation reduces the sensitivity of the output to distortions and shifts. Moreover, it allows the network to train faster, as also decreases the resolution of the feature maps (e.g. a 2x2 maxpooling layer takes the maximum value of 4 units, effectively dividing the number of neurons in a map by 4). In this thesis we only considered maxpooling layers.

We believe that the ability of CNNs to extract their own features is useful in financial time series classification as well, since creating hand-crafted features requires expert knowledge and is error prone. According to Zeng et al. (2014), CNN are ideally suited for feature extraction, and applying CNN to raw input data yields better results than e.g. first applying principal component analysis (PCA) to the data, as PCA only yields a linear combination of the raw features; CNN on the other hand are claimed to be able to capture the complex non-linear relationship, as they extract more meaningful features. Zeng et al. (2014) state that there are 2 main advantages to CNN for time series classification. First there is the fact that they capture local dependencies in the data. Similar to image pixels, technical indicators that are close to each other may go up or down at the same time, and the CNN is capable of extracting useful features. The second advantage is their scale invariance; the fluctuations in the data may be expressed in different numbers, but rather than depending on the technical indicators taking certain values, the CNN will be able to recognize the fluctuations behind them that are indicative of certain behaviour.

Gunduz et al. (2017) also cite several advantages of CNN for financial time series classification. Aside from the earlier mentioned advantage that they do not require hand-engineered features, eliminating human expertise, these authors argue that CNN are suitable for the problem as they have strong generalization, increasing their performance on out-of-period data, and are noise tolerant.

4.4.2 CNN tests and results

Similar to our LSTM test, we started by looking at the literature to see how other researchers use CNNs for various tasks. By adapting networks that work well for other tasks, we hope to get an insight into what would make a good architecture for our forex direction prediction problem.
Below is an overview of several (adapted) models we try. Given the success of Krizhevsky et al. (2012), who advocate the use of ReLU activations for CNN for classification, we will do so too. To facilitate the discussion on the different networks, we use the shorthand notation proposed by Pigou, Van Den Oord, Dieleman, Van Herreweghe & Dambre (2018): C(n_c) stands for a convolutional layer with n_c feature maps, P stands for a maxpooling layer, and D(n_d) denotes a feedforward layer with n_d neurons. S is the final layer, the softmax classifier. Unless otherwise stated, we use a kernel size of 5x5, as this seems to be most common. As mentioned earlier, we always use the Adam optimizer.

1. A first model was adapted from Ordóñez et al. (2016), who propose, among other methods, a CNN model with the following architecture: C(64)- C(64)- C(64)- C(64)-D(128)-D(128)-S. They argue that in their time series classification case, maxpooling layers are not necessary, as they have windowed data, limiting the downsampling possibility. Given our earlier experience with LSTM, and the tendency of the networks to quickly overfit, we attempted this network with 2 rather than 4 convolutional layers. With 50% dropout, this lead to a model with 50.73% test accuracy, and overfitted heavily. Increasing dropout to 80% and reducing the amount of neurons in the dense layers to 20 lead to a test accuracy of 51.83%.
2. A second model we tested was adapted from Babu et al. (2016). As they used it for RUL regression rather than classification, we adapted it to: C(8)-P-C(14)-P-D-S. As no information was given on the amount of neurons in the dense layer, we used 36. Again, the model overfitted quickly with a dropout of 50%, with 50.62% test accuracy. A dropout of 80% lead to a test accuracy of 51.17%.
3. Yang et al. (2015) propose the following CNN model for HAR: C(50)-P-C(40)-C(20)-D(400)-S. The third convolutional layer has a 3x3 kernel size, rather than 5x5. While they state that they do not use regularisation, we decide to apply 50% dropout, given the tendency of our networks to overfit. This lead to a test accuracy of 51.58%.
4. Next we tried the famous LeNet5 model (see also figure 4.6) presented by LeCun et al. (1998). The architecture is: C(6)-P-C(16)-P-C(120)-D(84)-S. We also apply a dropout of 50%, resulting in a test accuracy of 51.06%.
5. Inspired by LeNet5, we created a similar model. Following Ordóñez et al. (2016), we use no maxpooling layer: C(20)-C(40)-C(120)-D(36)-S, and 75% dropout. This model, however, quickly uses all available memory when training. To prevent this from happening, a maxpooling layer was added between the second and third convolutional layer: C(20)-C(40)-P-C(120)-D(36)-S. This model yielded a test accuracy of 52.67%.
6. Another model yielding 52.67% test accuracy is inspired on the model put forward in the tutorial by TensorFlow (2018). Our adapted model has the following architecture: C(32)-P-C(64)-P-D(10)-D(10), and 75% dropout chance.

7. In another attempt to create a network without maxpooling layers, we built the following CNN: C(32)-C(32)-D(10)-D(10), with 50% dropout. This model also had 52.67% test accuracy.

More models were trained, trying to exceed 52.67% test accuracy, with no success. Similar to our LSTM tests, we attempted to do a randomized grid search exploring different architectures. However, the RandomizedSearchCV method tended to claim all RAM when the first model was finished and the next one was started, completely bogging down the machine's performance. In order not to hinder other people's testing, it was considered prudent to not reattempt the search. Manually searching some 40 models, 3 times each seemed infeasible, as it is not possible to minimize downtime compared to an automated search. Thus, unfortunately, no search was performed.

As we did for our best LSTM model, we reran the highest performing models, mentioned in points 5, 6 & 7, to track how long it takes to train them, in terms of epochs and real time, as well as check their robustness. However, it turns out that these models are not nearly as robust as the LSTM or MLP models. In section 5.1 we compare these CNN models to the corresponding CLSTM models, and show that the CLSTM models are far more robust. We conclude that while it is possible to reach 52.67% test accuracy with CNN models, it is hard to reproduce. In the next section we will discuss the CLSTM architecture further, as well as some of the models we tested.

4.5 Convolutional long short-term memory network

4.5.1 Convolutional long short-term memory network architecture

A final model we created is the hybrid model between LSTM and convolutional neural networks. It combines the convolutional and optional pooling layers from a CNN with an LSTM network. The idea behind this is that this model is more suitable for time series classification than a traditional CNN, as it has LSTM recurrent layers which replace the dense layers, and are capable of capturing time dependencies (Hochreiter et al., 1997). We hypothesised that it also outperforms LSTM networks, because the convolutional and pooling layers are said to work as a feature extractor (LeCun et al., 2015), as explained in section 4.4.1. We believe these features will improve the

predictive performance of the LSTM network, which is heavily dependent on the quality of its input features (Zhao et al., 2017).

4.5.2 CLSTM tests and results

As we did for our CNN models, we started by trying several different CLSTM model architectures, adapted from publications that use CLSTM networks. This way we hope to get a feel for how the hyperparameters influence predictive performance, as well as how CNN and CLSTM networks compare. Some of the tested models are detailed below; the first 2 are adapted from the literature, and the last 3 are equivalent to models 5, 6 and 7, in section 4.4.2 on CNNs; we replaced the dense layers with LSTM layers, which allows us to compare predictive performance of both models. We use the same shorthand notation as for CNN, noting that $R(n_r)$ now denotes a layer of LSTM neurons, with n_r neurons.

1. The first model we tested was adapted from Ordóñez et al. (2016), who not only propose a CNN model for HAR, but also a CLSTM network. Their proposed CLSTM network is entirely similar to their CNN network, and is the following: C(64)- C(64)- C(64)- C(64)-R(128)-R(128)-S. Since this model's CNN equivalent is prone to using all memory capacity, we added maxpooling layers after the second and fourth convolutional layer, and use 32 neurons in each LSTM layer. We also apply 50% dropout between to the LSTM layers. This gives the following network: C(64)- C(64)-P- C(64)- C(64)-P-R(32)-R(32)-S. This yields a test accuracy of 52.67%.
2. Our second model was adapted from Zhao et al. (2017). For their RUL problem they propose the following architecture: C(150)-P-R(150)-R(200), using a 10x10 filter, and bidirectional LSTM layers. Given the fact that earlier models seemed prone to overfitting, we modified this into: C(50)-P-R(20)-R(20)-S, with 50% dropout and regular LSTM layers. This model resulted in 52.67% test accuracy.
3. The first model similar to a CNN we developed ourselves: C(20)-C(40)-P-C(120)-R(36)-S. With 50% dropout, this model achieves a test accuracy of 52.67%
4. Another model based on our own CNN is: C(32)-P-C(64)-P-R(10)-R(10)-S. This also results in 52.67% test accuracy.
5. The final model based on a CNN model is: C(32)-C(32)-R(10)-R(10)-S. Again, this model reaches 52.67% test accuracy, with 50% dropout.

Since we could not manage to perform a grid search for CNN, we also did not perform one for our CLSTM models. These test results, however, lead us to believe that the CLSTM might be able to achieve a test accuracy of just 52.67%. A grid search should be performed to confirm this. It should be noted that this is also the exact highest test accuracy we managed to achieve with CNN. A first conclusion seems to be that replacing the dense layers with LSTM layers does not add to the predictive power of the model. However, when running these models again, to measure how long it takes to train them and how robust they are, CLSTM networks appear to be far more robust. As mentioned earlier we will compare CNN and CLSTM networks for robustness and training time in section 5.1., where we find that CLSTM networks are far more robust.

5. Additional tests

So far we have discussed all of the models we applied to the problem of predicting whether foreign exchange rates will rise or fall. Looking at the results, we can establish that our initial hypothesis proves to be untrue: the hybrid CLSTM model does not outperform all other models. Some results were striking though, which is why we decided to explore these a little further in the sections below. First of all, it appears that CLSTM and CNN perform very similarly, with a test accuracy of 52.67%. This is discussed further in section 5.1, where we compare the 3 similar CLSTM and CNN networks. In section 5.2 we will discuss the different activation functions in more detail, and discuss how they influence performance of the LSTM and MLP network. This was already covered to some extent in section 4.3. In section 5.3 we apply our best performing networks on other currency pairs, to establish whether the results found earlier also hold for these pairs. We do this in two ways. First by retraining the models on these other currencies, to then test the models on those pairs. The second way is training the models on the EUR/USD currency pair for which they were optimised, and then test them on the other currency pairs. We believe this gives an idea of how good the models are at learning the patterns in the data rather than the data itself. We finish this section up by performing some statistical tests to see whether our models perform significantly better than random, and whether some models outperform others.

5.1 Comparison between CNN and CLSTM

Rather unexpectedly we found that both regular CNN as well as CLSTM manage to achieve a test accuracy of 52.67%, for several different model architectures. However, for CNN these results are hard to reproduce. In table 5.1 we summarize the results of training each of the three architectures for which we made CNN and CLSTM equivalents, and show how CNN are far less robust. We trained each architecture 3 times, and each run is denoted as the name of the model and the number of the run, e.g. CNN(1) is the first run for the CNN model. We still use the shorthand notation introduced earlier. X(n_x) now stands for a layer X, with n_x neurons. X is a dense layer in regular CNN, and an LSTM layer in CLSTM. Early stopping was used; the CNNs stopped after 3 epochs of increasing validation error, the CLSTM networks stopped after 2 worse iterations, as these are very robust. All models were trained with 50% dropout, except for the second and third CNN models, which performed better with 75% dropout.

Table 5. 1: Comparing CNN and CLSTM robustness

			CNN			CLSTM		
No	Architecture	Characteristics	CNN(1)	CNN(2)	CNN(3)	CLSTM(1)	CLSTM(2)	CLSTM(3)
1	C(32)-C(32)-X(10)-X(10)-S	**Test accuracy (%)**	**51,61**	**51,66**	**52,08**	**52,67**	**52,67**	**52,67**
		Training time (s)	35924	55477	53778	58658	78877	74415
		Number of epochs	3	3	3	2	2	2
2	C(32)-P-(64)-P-X(10)-X(10)-S	**Test accuracy (%)**	**51,91**	**52**	**52,52**	**52,67**	**52,67**	**52,67**
		Training time (s)	16048	22654	15538	20985	17036	19572
		Number of epochs	3	3	3	2	2	2
3	C(20)-C(40)-P-C(120)-X(36)-S	**Test accuracy (%)**	**52,13**	**51,01**	**50,93**	**52,68**	**52,67**	**52,67**
		Training time (s)	101741	37995	54295	80753	189922	141383
		Number of epochs	4	3	3	2	4	3

Indeed, this table confirms that for our forex direction prediction problem, CLSTM networks are far more robust than similar CNNs. Even though this comes with a larger training time, the increase in accuracy is preferable.

5.2 Comparison between MLP and LSTM

As discussed earlier, we find that the predictive accuracy of our MLP model heavily depends on the activation function used, more so than for our LSTM model. For our MLP, the activation function is the operation that is performed on the sum of the values that arrive at a neuron, as explained in section 4.1.1. In an LSTM network, activation functions are used at the several different gates. Originally, these functions were inspired on the functions used by neuroscientists to model the expected firing rate for a given input at synapses in the brain, such as the sigmoid and tanh function. According to Glorot et al. (2011), the tanh function tends to work better for MLPs than the sigmoid, but both are used frequently (Zheng, Liu, Chen, Ge & Zhao, 2014). The functions are shown in figure 5.1 below.

Figure 5. 1: Sigmoid and Tanh activation functions

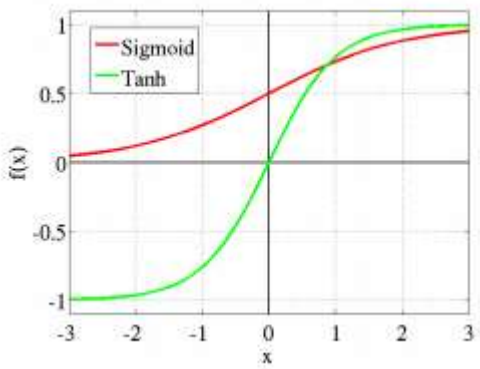


Image adapted from Glorot et al. (2011)

These functions, however, are non-saturating, which leads to longer training times, according to Krizhevsky et al. (2012). Glorot et al. (2011) argue propose a non-saturating function, the rectified linear unit (ReLU), as a higher performing alternative. They find that when using MLP for classification, using ReLU leads to a higher accuracy. This function is shown in figure 5.2 below.

Figure 5. 2: Rectifier linear units

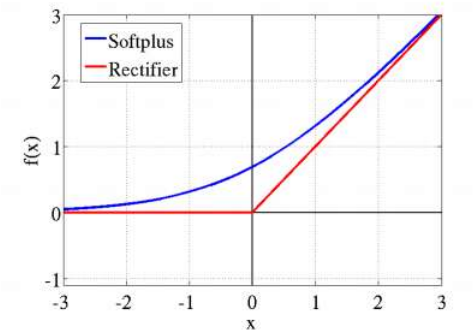


Image adapted from Glorot et al. (2011)

As explained in section 4.3, we find this as well, both in the literature consulted and in our own work. Note that we do not discuss the softplus activation in this thesis. Table 5.2 below compares predictive performance for the optimal LSTM network in function of different activation functions, with the performance of the MLP network, with a similar architecture and the same activation functions. These findings are visualized in figure 5.3 and 5.4, where we plot the average prediction accuracy over three runs, for both models in function of the activation function.

Table 5. 2: Comparing MLP and LSTM performance as a function of the activation function

			MLP			LSTM		
No	Activation function	Characteristics	MLP(1)	MLP(2)	MLP(3)	LSTM(1)	LSTM(2)	LSTM(3)
1	ReLU	**Test accuracy (%)**	**52,7**	**52,7**	**52,7**	**52,7**	**52,7**	**52,7**
		Training time (s)	472	394	371	9024	10920	11222
		Number of epochs	11	11	10	16	21	19
2	Sigmoid	**Test accuracy (%)**	**52,22**	**52,19**	**51,46**	**52,65**	**52,68**	**52,7**
		Training time (s)	370	490	928	3727	10135	8351
		Number of epochs	10	10	25	10	16	11
3	Tanh	**Test accuracy (%)**	**51,02**	**51,14**	**50,91**	**52,08**	**52,52**	**52,52**
		Training time (s)	486	403	405	5873	8303	7644
		Number of epochs	15	11	11	12	15	13

This table leads to two important conclusions: first of all, when using ReLU activations, MLP is as performant and robust as LSTM networks, in far less training time. In this case, there appears to be no advantage in using LSTM networks. Moreover, MLP also outperforms both CNN and CLSTM, both in terms of training time as predictive performance. As will be apparent from section 5.4, this

difference is not statistically significant, however. The second conclusion is, that both robustness and performance heavily decrease when using either sigmoid or tanh activation; this is true to a lesser extent for LSTM networks.

Figure 5. 3: average predictive accuracy in function of the activation function

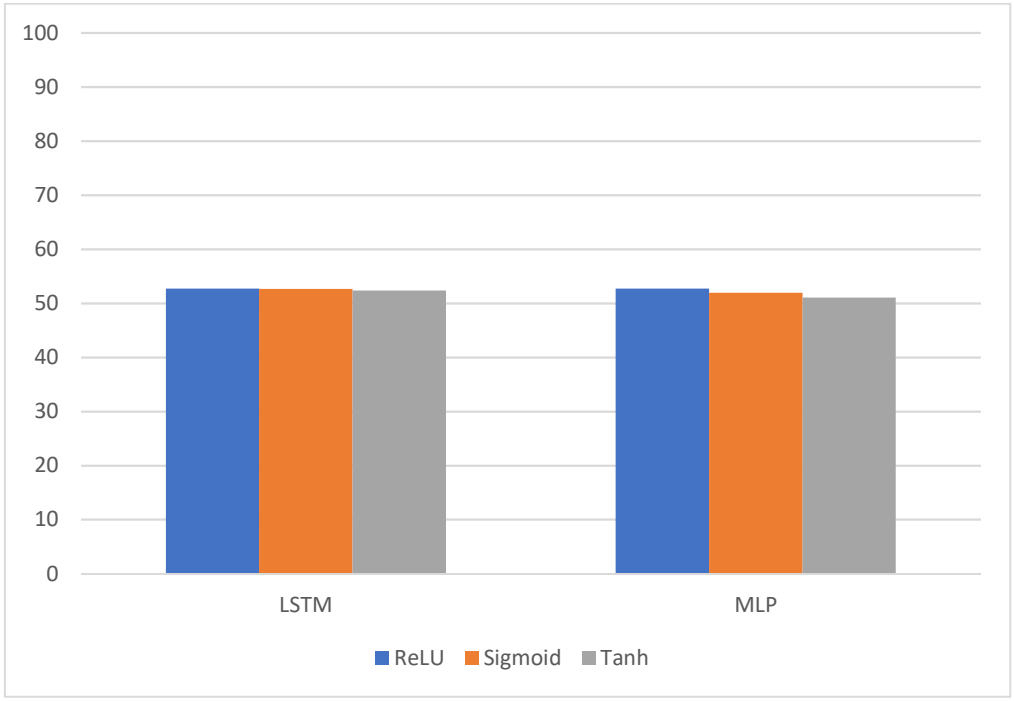
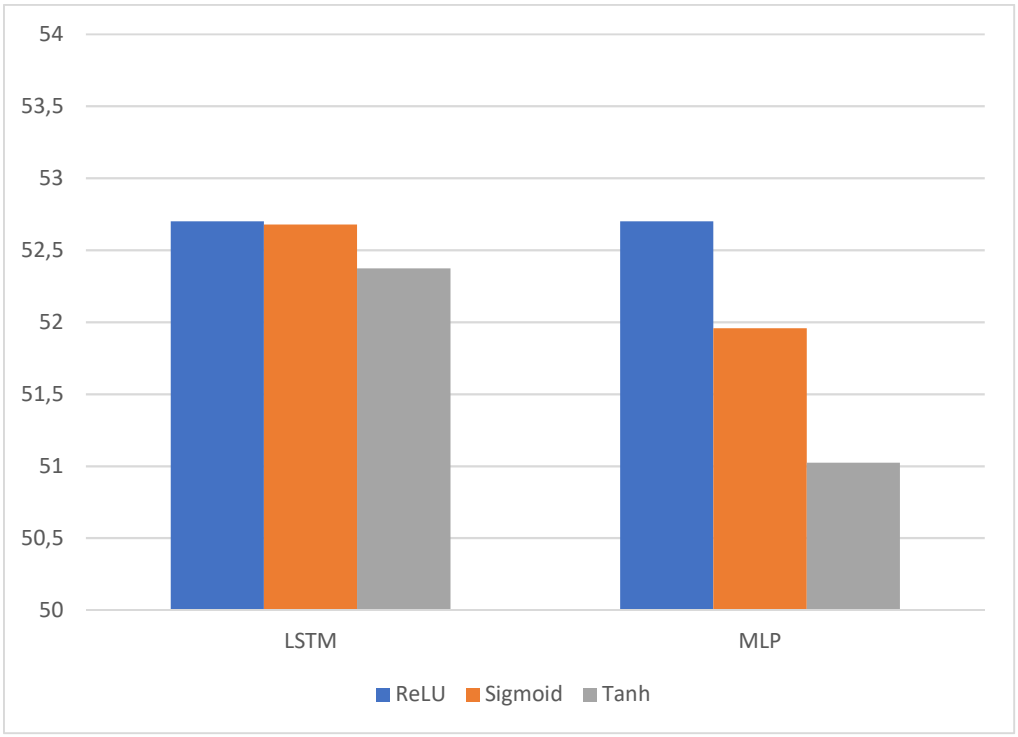


Figure 5. 4: Average predictive accuracy in function of the activation function (zoomed in)



5.3 Different currencies

In this section we apply the most successful architecture of each model to the data of two other currency pairs; these additional currency pairs are GBP/USD and USD/JPY. This seems interesting for two reasons. First, it allows us to see whether the models we optimised for the EUR/USD currency pair also work for other currencies. We will not try to optimise the architecture for these pairs, however. Second, it also functions as an additional validation of which models are best, and if the same patterns hold; i.e. LSTM and MLP prove to have the same predictive accuracy, and outperform CNN and CLSTM. The data is structured in the same way, again we use the data from 2012 until 2015 for training, the data from 2016 serves as the validation set, and the data from 2017 makes up the test set.

It turns out that these predictions can be made in two ways. We can train the models on the same pair that we try to predict, e.g. use 2012 until 2015 data of the GBP/USD currency pair to predict what will happen for the 2017 GBP/USD data. The second way is to use different pairs for training and validating on the one hand, and testing on the other. E.g. use the 2012-2015 EUR/USD data to make predictions for the 2017 GBP/USD data. According to LeCun et al. (2015), there should be no difference in predictive accuracy, as they argue that deep learning algorithms are capable of developing a certain "intuition" concerning the phenomenon they have to classify. They develop this intuition based on similar, but different observations, i.e. the training data, and manage to make good predictions for new data, the test set. We will explore both ways in this section.

5.3.1. Same training, validation and test currency pair

We start out by using data of the same currency pair for training, validating and testing. As mentioned, we will use the architectures that gave the best results for the EUR/USD currency pair for the predictions based on other data. These architectures are the following: for LSTM we use the architecture that was found to perform best in the randomized grid search. For MLP we used the adapted version from LSTM, both with ReLU activations, given our earlier findings. For CLSTM we decided to use the C(32)-P-C(64)-P-R(10)-R(10)-S model, which corresponds to model 2 in table 5.1. Given that all models performed equally well, we used the model that takes the least amount of time to train. We chose the corresponding equivalent model for CNN.

We ran each of the models only once on the data, assuming that LSTM, MLP and CLSTM will behave as robustly as before. We know that CNN might not deliver the best results. The test

accuracies achieved by each model for each currency pair are outlined in table 5.3 below. For ease of comparison, we add the results for the EUR/USD currency pair again.

Table 5. 3: Test accuracy for each model, trained on the different currency pairs

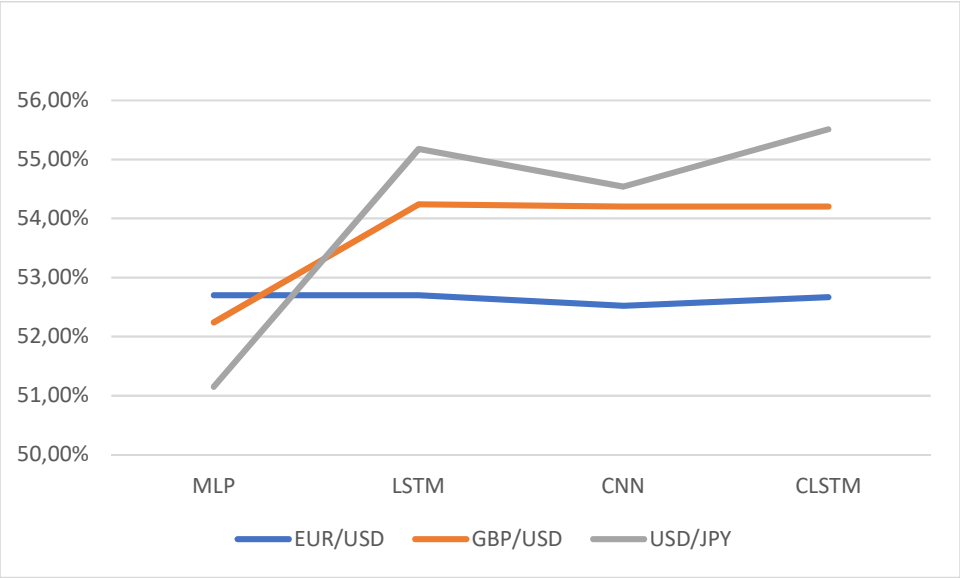
No	Currency pair	MLP	LSTM	CNN	CLSTM
1	EUR/USD	52,70%	52,70%	52,52%	52,67%
2	GBP/USD	52,24%	54,24%	54,20%	54,20%
3	USD/JPY	51,15%	55,18%	54,54%	55,51%

This table leads to some interesting findings. It is immediately clear that while MLP and LSTM have the same predictive performance for the EUR/USD currency pair, this does not hold for other currency pairs. We believe this is probably due to the fact that we optimised the LSTM network for the EUR/USD currency pair, and thus indirectly also optimised the MLP network, as we test its LSTM equivalent. When applied to other currency pairs, the simpler MLP model does not manage to perform as well as the other, more complex models. It appears that when optimised, and using ReLU activations, MLP is capable of performing equally well as the other models. The other models, however, are much more flexible; performing better on currency pairs for which they were not optimised.

Similar to the EUR/USD currency pair, the other networks display similar predictive behaviour for the GBP/USD pair: LSTM outperforms both CNN and CLSTM networks, although the difference is not that big, and in fact not statistically significant, as we will show in section 5.4.2. These 3 models also work very well for the USD/JPY currency pair, and here, as hypothesised, the CLSTM model even outperforms the others. This could be due to the fact that the EUR/USD and GBP/USD currency pairs might be more similar, as they are pairs between currencies of Western economies, and LSTM networks might be more suitable for these pairs, while CLSTM networks are more suitable for the USD/JPY currency pair. Perhaps different currencies call for different deep learning methods. This is only a hypothesis, however, and should be further investigated. This is beyond the scope of this thesis.

Figure 5.5 below visualises the test accuracy as a function of the different models and the different currency pairs. It appears that the accuracies for the different models are also a bit more diverse for the USD/JPY currency pair. Again, this graph is zoomed in for a nuanced interpretation.

Figure 5. 5: Predictive accuracy as function of currency pair & model (zoomed in)



These results naturally lead to the question of whether we would be able to achieve better results on the GBP/USD and USD/JPY currency pairs if we had optimised the models for these pairs. While this was not feasible due to time constraints, the results of the next section suggest that this is true.

5.3.2. Different training and test currency pair

As argued, our deep learning models should be capable of learning the patterns in the training data necessary to make good predictions on the test data. If we assume that the same patterns emerge in all currency data, and that all currency pairs behave in a similar way, we can hypothesise that our models should be able to make good predictions, even if they are trained on the data of a different currency pair. This is explored in this subsection. Table 5.4 below provides an overview of the test accuracy of our GBP/USD and USD/JPY currency pairs, for each model, and as a function of whether they are trained on the same currency pair, or the EUR/USD currency pair.

Table 5. 4: Test accuracy as a function of training data currency and model

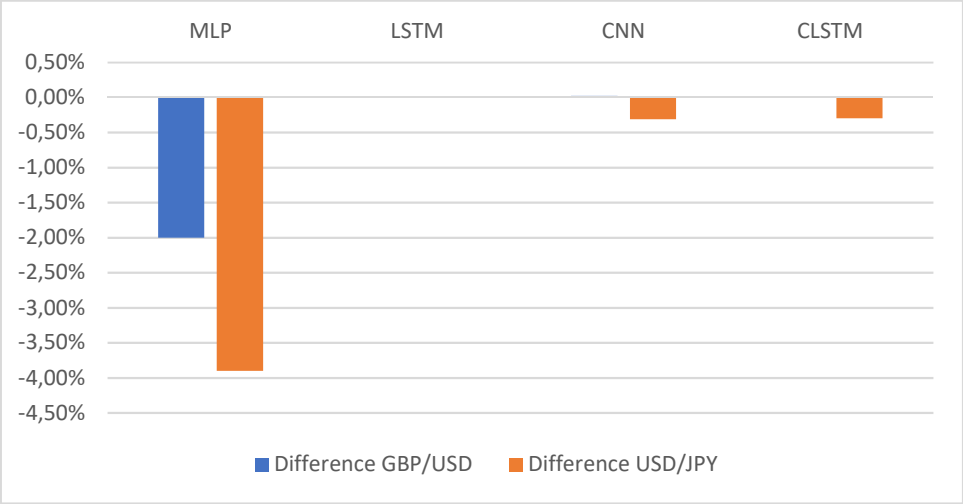
No	Currency pair	Trained on	MLP	LSTM	CNN	CLSTM
1	GBP/USD	EUR/USD	54,24%	**54,24%**	54,18%	**54,20%**
		GBP/USD	52,24%	**54,24%**	54,20%	**54,20%**
2	USD/JPY	EUR/USD	55,05%	**55,18%**	54,85%	55,51%
		USD/JPY	51,15%	**55,18%**	54,54%	55,21%

We indicated in bold for which models and currency pairs the test accuracy is the same for both model versions. Two important conclusions follow from this table. First of all, the LSTM network appears to work equally well for predicting test accuracy of a certain currency pair, regardless of whether it is trained for that currency pair or another.

Perhaps even more interesting is that the other models appear to often perform *better* when trained for the EUR/USD currency pair than when trained for the pair it is trying to predict. This is also visualised in figure 5.6 below, where we plot the difference in predictive accuracy of the model when trained for the currency pair it is tested on, as opposed to the accuracy achieved when trained on the EUR/USD currency pair. This leads to the conclusion that for our models, aside from the LSTM model, it could be more important that they are optimised for the currency pair that they are trained on, rather than that they are being trained on the currency pair they need to predict. This also supports our hypothesis that these models are indeed very capable of learning the patterns in the data, rather than the data itself, provided they are optimised for the task at hand. It is striking to see how well our models generalise to unseen data in the form of other currency pairs. While we did not test this, it would be interesting to determine the predictive accuracy of the models if we had optimised them, and trained them on these other currency pairs, and compared those test results with these, were we trained them on other data, for which they were optimised.

Training on the currency pair for which it is optimised rather than training on the pair it is trying to predict appears to be most important for the MLP network. While we find that it performs surprisingly well compared to the more complex models, its performance suffers most when unoptimized.

Figure 5. 6: Difference in predictive accuracy for training a model on the data it is tested on versus training on EUR/USD data



5.4 Tests of significance

In this section we will touch briefly on the statistical significance of our tests. Two types of tests can be made; first we will check whether our models are better at classifying if the exchange rate will go up or down than a random model. We will do this in subsection 5.4.1. Second, we will also look at whether certain models perform better than others. This will be elaborated in subsection 5.4.2. The statistical tests used in this section were adapted from Vincke (2014).

5.4.1 Significance with regard to a random model

It can be hypothesised that due to the law of large numbers, a random model used for binary classification will end up getting 50% of its guesses correct. We will determine the minimum accuracy a model needs to have to perform significantly better than such a random model.

Our problem, as a binary classification problem, will adhere to the Bernoulli distribution. The test set can then be thought of as a sample that is drawn from that Bernoulli distribution. An observation will be correctly classified with a probability equal to the test accuracy. The test set, consisting of the elements $X_{i,}$ is then a sample that follows the binomial distribution:

$$\sum_{i=1}^{n} X_i \sim B(n, p)$$

With n the amount of observations in the test set. Under the null hypothesis that the model does not significantly outperform a random model, $p \leq 0.50$. The central limit theorem allows us to assume that the binomial distribution will tend to the normal distribution. The sample mean, $\bar{X}$, is

then the test accuracy of the model. Under the central limit theorem we can assume that this will tend to the normal distribution:

$$\bar{X} = \hat{P} \sim N(p, \frac{p(1-p)}{p})$$

Thus, given the normal distribution, the test statistic becomes:

$$Z = \frac{\hat{P} - p_0}{\sqrt{\frac{p_0(1 - p_0)}{n}}} \overset{H_0}{\sim} N(0,1)$$

As can be learned from table 3.2, the EUR/USD currency pair has the smallest test set. We use $n = 2{,}239{,}771$ as it will provide the most stringent test. The lowest test accuracy achieved at any point in this thesis is 51.15% (see table 5.4). If we can determine that this accuracy is significant in combination with the smallest possible test set size, we can conclude that it is safe to assume that all test accuracies are significant. Filling in the numbers results in:

$$Z = \frac{0.5115 - 0.5}{\sqrt{\frac{0.5(1 - 0.5)}{2239771}}} = 34.42$$

Since $34.42 \geq 3.291$, we can safely assume that all of our models perform significantly better than random ($p < 0.0005$).

5.4.2 Comparative model performance

Rather than making an exhaustive list of whether a certain result is better than all other results, we will try to give an indication of how large the difference in predictive accuracy should be for it to be a significant difference. We will build on the assumption made in the previous subsection. Now we have two normally distributed sample means, and the difference between them will also be normally distributed. This gives the following test statistic:

$$Z = \frac{\widehat{P_1} - \widehat{P_2}}{\sqrt{\widehat{P_c}(1 - \widehat{P_c})}\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \overset{H_0}{\sim} N(0,1)$$

With $\widehat{P_c}$ the pooled sample proportion:

$$\widehat{P_c} = \frac{n_1\widehat{P_1} + n_2\widehat{P_2}}{n_1 + n_2}$$

As a simplification, we will say $n_1 = n_2 = 2{,}239{,}771$. $\widehat{P_c}$ then becomes:

$$\widehat{P_c} = \frac{\widehat{P_1} + \widehat{P_2}}{2}$$

If we want the difference between predictive accuracies of models to be significant for $p < 0.05$, then we need:

$$1.64 \leq \frac{\widehat{P_1} - \widehat{P_2}}{\sqrt{\widehat{P_c}(1 - \widehat{P_c})}\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

This only holds if:

$$\widehat{P_1} - \widehat{P_2} \geq 0.08$$

In other words, one model only significantly outperforms another if the difference in predictive accuracy between the two models exceeds 0.08 percentage points. This means that e.g. for the EUR/USD currency pair, LSTM and MLP did not significantly outperform the CNN and CLSTM models for $p < 0.05$.

6. Conclusion and limitations

In this thesis we applied several deep learning models to the problem of predicting whether foreign exchange rates will go up or down, on a per-minute basis, on out-of-sample data. The methods we used are convolutional and LSTM neural networks; a hybrid between the two, which we called CLSTM networks, and the more traditional MLP networks.

We showed that our first hypothesis, that this hybrid network would consistently outperform the other models, turned out to be untrue. We found that this hypothesis held for the USD/JPY currency pair, but not for the EUR/USD or GBP/USD currency pairs. For those two pairs the LSTM model performed best. We thus have to conclude that there is no model that consistently outperforms the others. It needs to be taken into account, however, that these model architectures were optimised for the EUR/USD currency pair. This is also what distinguishes the MLP model from the more complex models. It appears that when optimised, and using ReLU activations, MLP can perform equally well as the other, more complex models. The simpler MLP model, however, performs poorest when trained on currency data for which it was not optimised.

We showed that our second hypothesis does turn out to be true. The different deep learning methods are very capable of learning the relevant patterns in currency data, and generalising this to other currency pairs. When trained on a different currency pair than the pair it is trying to make a prediction for, all of our models still manage to make good predictions, that are statistically significant. Moreover, we found that in some cases it is even more important to train the models on the currency data for which they were optimised, rather than training them on the currency pair to which they will be applied; e.g. training a CLSTM model on the EUR/USD currency pair for which it was optimised will yield a higher out-of-sample accuracy on the USD/JPY test set than when training the same model architecture on the USD/JPY data, for which this architecture was not optimised.

Further ex post testing showed the importance of choosing the right activation function, when comparing MLP and LSTM networks for the data for which they were optimised. We showed that using the ReLU activation function can lead to MLP and LSTM networks performing equally well.

Another ex post test showed that while our CLSTM network was not consistently the best model, it was far more robust than its CNN counterpart.

While we discovered some interesting findings, several shortcomings in this thesis can be pointed out, providing possibilities for future research. The most important shortcoming lies in the fact that we did not test how models that are optimised for and trained on GBP/USD and USD/JPY data compare to models that are optimised for and trained on EUR/USD data, and then applied to GBP/USD and USD/JPY test data; as mentioned on section 5.3.2. Second, this thesis does not compare the different neural networks based on profitability. After all, investors are not only interested in whether FOREX will go up or down, but perhaps more in how much profit a certain prediction would allow them to make. It would be interesting to know how the different networks compare when using a trading model. Third, while the inputs include both historical data and technical indicators, no fundamental data was used. There exist several possible fundamental indicators that could be included, such as trade balance between the countries or political unions, GDP of the countries, employment rates, consumer price index, export & import volume, foreign reserve… Including these could improve the predictive accuracy of the models. Another interesting additional type of input data comes in the form of textual data. Several sources successfully include data from financial news articles, which leads to higher predictive performance. As argued by Vargas et al. (2017), financial news articles often contain important information that influences the price or rate of a financial product, and this information is not necessarily captured in historical data.

Further research could also go deeper into the models and their parameters presented in this thesis. For example, we based the size of the time window on what seems common in the literature, but we have not tried to determine an optimal window size. We only performed a grid search for LSTM networks, and failed to do so for CNN and CLSTM networks. Another avenue for future research lies in the application of CLSTM networks to other classification problems. While we did not find that this network consistently outperforms others, we do prove that they are far more robust than their CNN counterparts, at a similar training time. It would be interesting to see whether CLSTM networks can provide a more robust alternative to CNNs in other classification tasks as well.

Similar research could be performed for MLP networks. Our findings confirm the results of Glorot et al. (2011), that the performance of MLP networks can be heavily improved by using ReLU activations. In this thesis, we find that this also holds true for financial time series classification, performing equally well as LSTM networks. It could be further investigated how LSTM and MLP compare in other time series forecasting tasks, when using ReLU activations.

Refences

A guide to TF layers: building a convolutional neural network. (2018). Retrieved from TensorFlow: https://www.tensorflow.org/tutorials/layers

Akita, R., Yoshihara, A., Matsubara, T., & Uehara, K. (2016). Deep learning for stock prediction using numerical and textual information. *Computer and information science (ICIS), 2016 IEEE/ACIS 15th international conference*, 1-6.

Armano, G., Marchesi, M., & Murru, A. (2005). A hybrid genetic-neural architecture for stock indexes forecasting. *Information Sciences, 170 (1)*, 3-33.

Atsalakis, G., & Valavanis, K. (2009). Surveying stock market forecasting techniques - part II: soft computing methods. *Expert systems with applications 36(3)*, 5932-5941.

Babu, G., Zhao, P., & Li, X. (2016). Deep convolutional neural network based regression approach for estimation of remaining useful life. *International conference on database systems for advanced applications* (pp. 214-228). Springer, Cham.

Ballings, M., Van den Poel, D., Hespeels, N., & Gryp, R. (2015). Evaluating multiple classifiers for stock price direction prediction. *Expert systems with applications 42(20)*, 7046-7056.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of machine learning research, 13(Feb)*, 281-305.

Box, G., & Jenkins, G. (1970). *Time series analysis, forecasting, and control.* Francisco Holden-Day.

Chao, J., Shen, F., & Zhao, J. (2011, July). Forecasting exchange rate with deep belief networks. *Neural networks (IJCNN), The 2011 international joint conference* (pp. 1259-1266). IEEEE.

Chen, K., Zhou, Y., & Dai, F. (2015, October). A LSTM-based method for stock returns prediction: a case study of China stock market. *Big Data (Big Data), 2015 IEEE international conference* (pp. 2823-2824). IEEE.

De Gooijer, J. G., & Hyndman, R. J. (2006). 25 years of time series forecasting. *International journal of forecasting 22(3)*, 443-473.

Di Persio, L., & Honchar, O. (2016). Artificial neural networks approach to the forecast of stock market price movements. *International journal of economics and managemt systems, 1*, 158-162.

Eck, D., & Schmidhuber, J. (2002, September). Finding temporal structure in music: Blues improvisation with LSTM recurrent networks. *Neural networks for signal processing, 2002. Proceedings of the 2002 12th IEEE workshop* (pp. 747-756). IEEE.

ElSaid, A., Wild, B., Higgins, J., & Desell, T. (2016, October). Using LSTM recurrent neural networks to predict excess vibration events in aircraft engines. *e-Science (e-Science), 2016 IEEE 12th international conference* (pp. 260-269). IEEE.

Fama, E. F. (1970). Efficient capital markets: a review of theory and empirical work. *The journal of finance, 25(2)*, 383-417.

Galeshchuk, S., & Mukherjee, S. (2017). Deep networks for predicting direction of change in foreign exchange rates. *Intelligent systems in accounting, finance and management, 24(4)*, 100-110.

German, S., Bienenstock, E., & Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural computation, 4(1)*, 1-58.

Géron, A. (2017). *Hands-on machine learning with scikit-learn and tensorflow.* California, United States: O'Reilly Media, Inc.

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999). Learning to forget: continual prediction with LSTM. 850-855.

Glantz, M., & Kissell, R. (2013). *Multi-asset risk modelling: techniques for a global economy in an electronic and algorithmic trading era.* Academic Press.

Glorot, X., Bordes, A., & Bengio, Y. (2011, June). Deep sparse rectifier networks. *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, (pp. 315-323).

Graves, A., Mohamed, A. R., & Hinton, G. (2013, May). Speech recognition with deep recurrent neural networks. *Acoustiscs, speech and signal processing (icassp), 2013 ieee international conference* (pp. 6645-6649). IEEE.

Gunduz, H., Yaslan, Y., & Cataltepe, Z. (2017). Intraday prediction of Borsa Istanbul using convolutional neural networks and feature correlations. *Knowledge-based systems, 137*, 138-148.

Guresen, E., Kayakutlu, G., & Daim, T. (2011). Using artificial neural networks models in stock market index prediction. *Expert systems with applications 38(8)*, 10389-10397.

Hinton, G., Srivastava , N., Krizhevsky, A., Sutskever, I., & Slakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580.*

Historical data feed. (2018). Retrieved from dukascopy.com: https://www.dukascopy.com/swiss/english/marketwatch/historical/

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International journal of uncertainty, fuzziness and knowledge-based systems 6(02)*, 107-116.

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

Jeremy, J. (2018, March 01). *Setting the learning rate of your neural network*. Retrieved from Jeremy Jordan: https://www.jeremyjordan.me/nn-learning-rate/

Kamruzzaman, J., & Sarker, R. A. (2003, December). Forecasting of currency exchange rates using ANN: a case study. *Neural networks and signal processing, 2003. Proceedings of the 2003 international conference* (pp. 797-797). IEEE.

Kayal, A. (2010). A neural networks filtering mechanism for foreign exchange trading signals. *Intelligent computing and intelligent systems (ICIS), 2010 IEEE international conference* (pp. 159-167). IEEE.

Kingma, D., & Ba, J. (2014). Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980.*

Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. *IJCAI, 14(2)*, 1137-1145.

Krizhevsky, A., Sutskever, I., & Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information precessing systems*, (pp. 1097-1105).

LeCun, Y., & Bengio , Y. (1995). Convolutional networks for images, speech and time series. *The handbook of brain theory and neural networks, 3361(10)*.

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

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86(11)*, 2278-2324.

Lipton, Z., Kale, D., & Wetzell, R. (2017). Phenotyping of clinical time series with lstm recurrent neural networks. *arXiv preprint arXiv:1510.07641.*

Ma, X., Tao, Z., Wang, Y., Yu, H., & Wang, Y. (2015). Long short-term memory neural network for traffic speed predcition using remote microwave sensor data. *Transportation Research part C: Emerging technologies, 54*, 187-197.

Malkiel, B. (1973). *A random walk down wall street.* United States: W. W. Norton & Company, Inc.

Malkiel, B. G. (2003). The efficient market hypothesis and its critics. *Journal of economic perspectives 17 (1)*, 59-82.

Martinez, L. C., da Hora, D. N., Palotti, J. R., Meira, W., & Pappa, G. L. (2009, June). From an artificial neural network to a stock market day-trading system: a case study on the BM&F BOVESPA. *Neural networks, 2009. IJCNN 2009. International joint conference* (pp. 2006-2013). IEEE.

Nelson, D., Pereira , A., & de Oliveira, R. (2017, May). Stock market's price movement prediction with LSTM neural networks. *Neural networks (IJCNN), 2017 international joint conference* (pp. 1419-1426). IEEE.

Ordóñez, F., & Roggen, D. (2016). Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition. *Sensors, 16(1)*, 115.

Pasero, E., Raimondo, G., & Ruffa, S. (2010). MULP: a multi-layer perceptron application to long-term, out-of-sample time series prediction. *International Symposium on neural networks* (pp. 566-575). Springer, Berlin, Heidelberg.

Patel, J., Shah, S., Thakkar, P., & Kotecha, K. (2015). Predicting stock and stock price index movement using trend deterministic data preparation and machine learning techniques. *Expert systems with applications 42*, 259-268.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. *Journal of machine learning research, 12(Oct)*, 2825-2830.

Pigou, L., Van Den Oord, A., Dieleman , S., Van Herreweghe, M., & Dambre, J. (2018). Beyond temporal pooling: recurrence and temporal convolutions for gesture recognition in video. *International journal of computer vision, 126(2-4)*, 430-439.

Prechelt, L. (1998). Early stopping-but when? In L. Prechelt, *Neural networks: tricks of the trade* (pp. 55-69). Berlin, Heidelberg: Springer.

Rumelhart, D., Hinton, G., & Williams, R. (1986). Learning representations by back-propagating errors. *Nature, 323(6088)*, 533.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research, 15(1)*, 1929-1958.

Technical indicators and overlays. (2018). Retrieved from StockCharts: http://stockcharts.com/school/doku.php?id=chart_school:technical_indicators

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

Triennial central bank survey. (2016). pp. 1-23.

Vanstone, B., & Finnie, G. (2009). An empirical methodology for developing stockmarket trading systems using artificial neural network. *Expert systems with applications, 36*, 6668-6680.

Vargas, M., de Lima, B., & Evsukoff, A. (2017, June). Deep learning for stock market prediction from financial news articles. *Computational intelligence and virtual environments for measurement systems and applications (CIVEMSA), 2017 IEEE international conference* (pp. 60-65). IEEE.

Vincke, D. (2014). *Toegepaste Statistiek II(a).* Gent.

White, H. (1988). Economic prediction using neural networks: the case of IBM stock returns. 451-458.

Yang, J., Nguyen , M., San, P., Li, X., & Krishnaswamy, S. (2015). Deep convolutional neural networks on multichannel time series for human activity recognition . *IJCAI*, 3995-4001.

Yao, J., & Lim Tan, C. (2000). A case study on using neural networks to perform technical forecasting of forex. *Neurocomputing 34*, (pp. 79-98).

Yuan, M., Wu, Y., & Lin, L. (2016, October). Fault diagnosis and remaining useful life estimation of aero engine using LSTM neural network. *Aircraft utility systems (AUS), IEEE international conference* (pp. 135-140). IEEE.

Zeiler, M. (2012). ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Zeng, M., Nguyen, L., Yu, B., Mengshoel, O., Zhu, J., Wu , P., & Zhang, J. (2014, November). Convolutional neural networks for human activity recgnition using mobile sensors. *Mobile computing, applications and services (MobiCASE), 2014 6th international conference* (pp. 197-205). IEEE.

Zhao, R., Yan, R., Wang, J., & Mao, K. (2017). Learning to monitor machine health with convolutional bi-directional lstm networks. *Sensors 17(2)*, 273.

Zheng, Y., Liu, Q., Chen, E., Yong, G., & Zhao, J. (2014, June). Time series classification using multi-channels deep convolutional neural networks. *International conference on web-age information management* (pp. 298-310). Springer, Cham.

Appendix

Appendix A: Manual to the code

While it is not possible to discuss all of the code line-by-line, in full detail, we will attempt to give a clear and structured overview on how to run all of the code in the next appendices, what order to run it in, what datasets each of the scripts produce… we will also point to topics of interest in the code, where relevant. If you attempt to run the code yourself, make sure to change the paths of the files to where you have them stored, and change the names of the files to the currency pair you want to check. The zip that we handed in online does not contain all of the datasets that were created at any given point, but it does contain the most important ones: the raw data, the data after we created all variables, and all of the time windows. These time windows can be used immediately for deep learning; it is thus not necessary to run any of the code that pre-processes or creates data.

Reading in and preparing data

In the zip that was handed in online, all of the code and the different datasets are included as well. We included all of the raw datasets, downloaded from dukascopy. Before downloading, we made sure to filter all the flat periods. The first script uploaded as dataprocessing.py and listed in appendix B loads the 6 different datasets for a currency, one per year, drops the volume column and attaches them together. This larger dataset is then saved to CSV; with a name similar to data12_17_EUR_USD.csv

Variable creation #1

The script variable_creation1.py creates the first set of additional variables, as well as the dependent variable. As input it takes the CSV generated by the last python script. All the code should be self-explanatory. It saves the data to a csv with a name similar to basetable_final_USD_JPY.csv. The code is uploaded, and attached in appendix C. Running this script can take 30 hours. At each created variable an intermediate dataset is written to csv, as a safety measure. These datasets could not be uploaded, due to size constraints.

Variable creation #2

This script creates the second half of the variables, and works very similar to the last script. It takes the CSV file from the last script as input, and outputs a file with a name similar to basetable_final_USD_JPY_v2.csv. This data set was handed in. Note that for these indicators we used the TALIB library, and it runs in just a couple of minutes. The code is added in appendix D.

Normalisation

When the variables are created, it is time to normalise them. This is done in script normalisation.py, which again takes the CSV from the last step, and saves the data to a CSV with a name similar to basetable_normalized_USD_JPY.csv. The code is added in appendix E. The normalized data is not added, due to size constraints. This script can be run in a very short amount of time. The data produced in the next step is the data that is actually fed into the deep learning algorithms, and this is added.

Data windowing

Now that the data is fully prepared, we can start making our time windows. In order to fully use the 33% computing power that was assigned to us, we made 8 scripts, that each make windows out of a certain part of the data. These windows are then written to .npy files, which are read in when a model is trained. Because the time to create the windows increases as the .npy files grow larger, we decided to split it up in 90 .npy files for the segments, and 90 .npy files for the corresponding labels. Each of these files contain 25.000 time windows. The .npy files are written away, and have a name similar to labels1.npy or segments1.npy. They are all included in the zip handed in online. All of the scripts first read in the csv produced by the last script. We included only one of these scripts in appendix F, as the others are entirely similar. This script is called window1.py. All of the windowing scripts (window1.py through window8.py) are uploaded online.

Deep learning code

We created a large number of different models for this thesis; it seems unnecessary to add them all. One should be able to fairly easily create all other models based on the code that we did add. We included the following scripts both in the zip handed in online and in the following appendices:

- Appendix G contains the script MLP.py, which runs the MLP model. Note that in the very beginning this script reads in all of the segments and the labels. **All of the deep learning scripts do this, but we only include that part of the code for the MLP.py script, to save space.**
- Appendix H contains the script LSTM.py. This is the model that we found to the best configuration, through our randomised grid search.
- Appendix I contains the script CNN.py, with the code to the model that trained fastest in our tests. It corresponds to model 2 in table 5.1.
- Appendix J contains the script CLSTM.py, which is the model that corresponds to the CNN model in appendix I.
- Appendix K contains the script LSTM_grid_search.py. This is the code that we ran to perform the randomised grid search for the LSTM model. Note that it is important to leave the seed as it is, as it allows to recreate the results.

Appendix B: Data preprocessing

```
import numpy as np
import pandas as pd

data12 = pd.read_csv("C:/Users/Simon/Google Drive/THESIS/Materiaal thesis (1)/FOREX DATA/USD_JPY12.csv")
data13 = pd.read_csv("C:/Users/Simon/Google Drive/THESIS/Materiaal thesis (1)/FOREX DATA/USD_JPY13.csv")
data14 = pd.read_csv("C:/Users/Simon/Google Drive/THESIS/Materiaal thesis (1)/FOREX DATA/USD_JPY14.csv")
data15 = pd.read_csv("C:/Users/Simon/Google Drive/THESIS/Materiaal thesis (1)/FOREX DATA/USD_JPY15.csv")
data16 = pd.read_csv("C:/Users/Simon/Google Drive/THESIS/Materiaal thesis (1)/FOREX DATA/USD_JPY16.csv")
data17 = pd.read_csv("C:/Users/Simon/Google Drive/THESIS/Materiaal thesis (1)/FOREX DATA/USD_JPY17.csv")

data = data12.append(data13)
```

```
data = data.append(data14)
data = data.append(data15)
data = data.append(data16)
data = data.append(data17)

#We will drop the volume column, as the traded volume for currencies can not be accurately measured.
data = data.drop(data.columns[[0,5]], axis = 1)

#write this to csv
data.to_csv("data12_17_USD_JPY.csv")
```

Appendix C: First variable creation

```
# #  variable creation
import pandas as pd
import numpy as np
data = pd.read_csv("E:/Thesis_Simon_Serrarens/Data/data12_17_USD_JPY.csv") #Fill in the appropriate path here


#When reading in data, a new column with rownumbers is automatically created. we need to drop this
data = data.drop(data.columns[[0]], axis = 1)

#First we create the moving average of the closing price, for 3 different time lags: 5, 10, 20
#Note that we also create the 12 and 26 minute MA, as we will use this for the MACD
data["MA_5"] = 0
data["MA_10"] = 0
data["MA_12"] = 0
data["MA_20"] = 0
data["MA_26"] = 0
```

```
for i in range(25, data.shape[0]): #Start with the 26th row, because you need the 25 preceding rows to base the value on
    count = 0
    if i % 100000 == 0:
            count +=1
            print(count*100000)

        #To create MA_5:
    total_5 = (data.loc[i-4, "Close"] + data.loc[i-3, "Close"] + data.loc[i-2, "Close"]
        + data.loc[i-1, "Close"] + data.loc[i, "Close"])
    data.loc[i, "MA_5"] = total_5/5


    #To create MA_10:
    total_10 = (data.loc[i-9, "Close"] + data.loc[i-8, "Close"] + data.loc[i-7, "Close"]
            + data.loc[i-6, "Close"] + data.loc[i-5, "Close"]
            + total_5)
    data.loc[i, "MA_10"] = total_10/10


    #To create MA_12:
    total_12 = (data.loc[i-11, "Close"] + data.loc[i-10, "Close"] + total_10)
    data.loc[i, "MA_12"] = total_12/12


    #To create MA_20:
    total_20 = (data.loc[i-19, "Close"] + data.loc[i-18, "Close"] + data.loc[i-17, "Close"]
            + data.loc[i-16, "Close"] + data.loc[i-15, "Close"]
            + data.loc[i-14, "Close"] + data.loc[i-13, "Close"] + data.loc[i-12, "Close"]
            + total_12)
    data.loc[i, "MA_20"] = total_20/20
```

```
    #To create MA_26:
    total_26 = (data.loc[i-25, "Close"] + data.loc[i-24, "Close"] + data.loc[i-23, "Close"] + data.loc[i-22, "Close"]
                + data.loc[i-21, "Close"]+ data.loc[i-20, "Close"] + total_20)
    data.loc[i, "MA_26"] = total_26/26
print("De moving averages zijn klaar.")


# In[ ]:


#Let's already save this file, and start from here. This way we don't need to always do all steps.
basetable_MA = data
basetable_MA.to_csv("basetable_MA.csv")

#Again we will already include the EMA_12 and EMA_26 for the MACD later on
basetable_MA["EMA_5"] = 0
basetable_MA["EMA_10"] = 0
basetable_MA["EMA_12"] = 0
basetable_MA["EMA_20"] = 0
basetable_MA["EMA_26"] = 0
#Initialise the first value of EMA_5, EMA_10 and EMA_20 by assigning them the value of their respective MA
basetable_MA.loc[25, "EMA_5"] = basetable_MA.loc[25, "MA_5"]
basetable_MA.loc[25, "EMA_10"] = basetable_MA.loc[25, "MA_10"]
basetable_MA.loc[25, "EMA_12"] = basetable_MA.loc[25, "MA_12"]
basetable_MA.loc[25, "EMA_20"] = basetable_MA.loc[25, "MA_20"]
basetable_MA.loc[25, "EMA_26"] = basetable_MA.loc[25, "MA_26"]
#Create the multipliers
multiplier_5 = (2 / 6)
multiplier_10 = (2 / 11)
multiplier_12 = (2 / 13)
```

```
multiplier_20 = (2 / 21)
multiplier_26 = (2 / 27)
for i in range(26, basetable_MA.shape[0]): #We start with the 27th row, as we initialised the EMA of the 26th row

        #To create EMA_5
    basetable_MA.loc[i, "EMA_5"] = ((basetable_MA.loc[i, "Close"] - basetable_MA.loc[i-1, "EMA_5"])*multiplier_5
                    + basetable_MA.loc[i-1, "EMA_5"])

    #To create EMA_10
    basetable_MA.loc[i, "EMA_10"] = ((basetable_MA.loc[i, "Close"] - basetable_MA.loc[i-1, "EMA_10"])*multiplier_10
                    + basetable_MA.loc[i-1, "EMA_10"])

    #To create EMA_12
    basetable_MA.loc[i, "EMA_12"] = ((basetable_MA.loc[i, "Close"] - basetable_MA.loc[i-1, "EMA_12"])*multiplier_12
                    + basetable_MA.loc[i-1, "EMA_12"])

    #To create EMA_20
    basetable_MA.loc[i, "EMA_20"] = ((basetable_MA.loc[i, "Close"] - basetable_MA.loc[i-1, "EMA_20"])*multiplier_20
                    + basetable_MA.loc[i-1, "EMA_20"])

    #To create EMA_26
    basetable_MA.loc[i, "EMA_26"] = ((basetable_MA.loc[i, "Close"] - basetable_MA.loc[i-1, "EMA_26"])*multiplier_26
                    + basetable_MA.loc[i-1, "EMA_26"])


# In[ ]:
```

```
basetable_EMA = basetable_MA
#Let's save it to file, then we can always pick it up from here
basetable_EMA.to_csv("basetable_EMA.csv")
#From this point on we should move back to this file, as it takes a while to compute the MA and EMA


# In[ ]:


#Again we start with the 26th row, since the ones before aren't complete and will be filtered out
basetable_EMA["MACD"] = 0
for i in range(25, basetable_EMA.shape[0]):
    basetable_EMA.loc[i, "MACD"] = basetable_EMA.loc[i, "EMA_12"] - basetable_EMA.loc[i, "EMA_26"]


# In[ ]:


#Let's drop the MA12, MA26, EMA12 and EMA26, and save this file.
basetable_MACD = basetable_EMA
basetable_MACD = basetable_MACD.drop(basetable_MACD.columns[[6, 8, 11, 13]], axis = 1)
basetable_MACD.to_csv("basetable_MACD.csv")


# In[ ]:


basetable_MACD["%R"] = 0
basetable_MACD["highest"] = 0
basetable_MACD["lowest"] = 0
for i in range(25, basetable_MACD.shape[0]):
```

```
    basetable_MACD.loc[i, "highest"] = max(basetable_MACD.loc[i-13, "Close"], basetable_MACD.loc[i-12, "Close"],
                                           basetable_MACD.loc[i-11, "Close"], basetable_MACD.loc[i-10, "Close"],
                                           basetable_MACD.loc[i-9, "Close"], basetable_MACD.loc[i-8, "Close"],
                                           basetable_MACD.loc[i-7, "Close"], basetable_MACD.loc[i-6, "Close"],
                                           basetable_MACD.loc[i-5, "Close"], basetable_MACD.loc[i-4, "Close"],
                                           basetable_MACD.loc[i-3, "Close"] ,basetable_MACD.loc[i-2, "Close"],
                                           basetable_MACD.loc[i-1, "Close"] ,basetable_MACD.loc[i, "Close"])

    basetable_MACD.loc[i, "lowest"] = min(basetable_MACD.loc[i-13, "Close"], basetable_MACD.loc[i-12, "Close"],
                                           basetable_MACD.loc[i-11, "Close"], basetable_MACD.loc[i-10, "Close"],
                                           basetable_MACD.loc[i-9, "Close"], basetable_MACD.loc[i-8, "Close"],
                                           basetable_MACD.loc[i-7, "Close"], basetable_MACD.loc[i-6, "Close"],
                                           basetable_MACD.loc[i-5, "Close"], basetable_MACD.loc[i-4, "Close"],
                                           basetable_MACD.loc[i-3, "Close"] ,basetable_MACD.loc[i-2, "Close"],
                                           basetable_MACD.loc[i-1, "Close"] ,basetable_MACD.loc[i, "Close"])

    basetable_MACD.loc[i, "%R"] = ((basetable_MACD.loc[i, "highest"] - basetable_MACD.loc[i, "Close"])
                                   / (basetable_MACD.loc[i, "highest"] - basetable_MACD.loc[i, "lowest"])) *(-100)


# In[ ]:


basetable_MACD = basetable_MACD.fillna(method='ffill')
basetable_MACD.describe()


# In[ ]:
```

```
#Finally we drop the columns we don't need (highest and lowest) and save the new intermediate
basetable
basetable_MACD = basetable_MACD.drop(basetable_MACD.columns[[12, 13]], axis = 1)
basetable_MACD


basetable_R = basetable_MACD
basetable_R.to_csv("basetable_R.csv")


#Next up: stochastic oscillator
#Again the default period is 14 days. This code is very similar to %R
basetable_R["%K"] = 0
basetable_R["highest"] = 0
basetable_R["lowest"] = 0
for i in range(25, basetable_R.shape[0]):
   basetable_R.loc[i, "highest"] = max(basetable_R.loc[i-13, "Close"], basetable_R.loc[i-12,
"Close"],
                                       basetable_R.loc[i-11, "Close"], basetable_R.loc[i-10, "Close"],
                                       basetable_R.loc[i-9, "Close"], basetable_R.loc[i-8, "Close"],
                                       basetable_R.loc[i-7, "Close"], basetable_R.loc[i-6, "Close"],
                                       basetable_R.loc[i-5, "Close"], basetable_R.loc[i-4, "Close"],
                                       basetable_R.loc[i-3, "Close"] ,basetable_R.loc[i-2, "Close"],
                                       basetable_R.loc[i-1, "Close"] ,basetable_R.loc[i, "Close"])

   basetable_R.loc[i, "lowest"] = min(basetable_R.loc[i-13, "Close"], basetable_R.loc[i-12, "Close"],
                                       basetable_R.loc[i-11, "Close"], basetable_R.loc[i-10, "Close"],
                                       basetable_R.loc[i-9, "Close"], basetable_R.loc[i-8, "Close"],
                                       basetable_R.loc[i-7, "Close"], basetable_R.loc[i-6, "Close"],
                                       basetable_R.loc[i-5, "Close"], basetable_R.loc[i-4, "Close"],
                                       basetable_R.loc[i-3, "Close"] ,basetable_R.loc[i-2, "Close"],
                                       basetable_R.loc[i-1, "Close"] ,basetable_R.loc[i, "Close"])
```

```
    basetable_R.loc[i, "%K"] = ((basetable_R.loc[i, "Close"] - basetable_R.loc[i, "lowest"])
                        / (basetable_R.loc[i, "highest"] - basetable_R.loc[i, "lowest"]) * 100)


# In[ ]:

basetable_R = basetable_R.drop(basetable_R.columns[[13,14]], axis = 1)
basetable_K = basetable_R


# In[ ]:

basetable_K = basetable_K.fillna(method='ffill')


# In[ ]:

basetable_K.to_csv("basetable_K.csv")


# In[ ]:

basetable_K["TP"] = 0        #Typical price
basetable_K["CCI"] = 0       #Commodity channel index
basetable_K["SMA_TP"] = 0    #20 minute standard moving average of TP
basetable_K["MD"] = 0        #Mean Deviation of TP
#we will need to create the typical price for the first rows as well, in order to be able to calculate
the moving
#average as of the 26th row:
for i in range(0, 25):
```

```
    basetable_K.loc[i, "TP"] = (basetable_K.loc[i, "High"] + basetable_K.loc[i, "Low"] +
basetable_K.loc[i, "Close"])/3
for i in range(25, basetable_K.shape[0]):
    #Typical price:
    basetable_K.loc[i, "TP"] = (basetable_K.loc[i, "High"] + basetable_K.loc[i, "Low"] +
basetable_K.loc[i, "Close"])/3

    #20 minute standard moving average of TP:
    basetable_K.loc[i, "SMA_TP"] = (basetable_K.loc[i-19, "TP"] + basetable_K.loc[i-18, "TP"] +
basetable_K.loc[i-17, "TP"]
                        + basetable_K.loc[i-16, "TP"] + basetable_K.loc[i-15, "TP"] + basetable_K.loc[i-
14, "TP"]
                        + basetable_K.loc[i-13, "TP"] + basetable_K.loc[i-12, "TP"] + basetable_K.loc[i-
11, "TP"]
                        + basetable_K.loc[i-10, "TP"] + basetable_K.loc[i-9, "TP"] + basetable_K.loc[i-
8, "TP"]
                        + basetable_K.loc[i-7, "TP"] + basetable_K.loc[i-6, "TP"] + basetable_K.loc[i-5,
"TP"]
                        + basetable_K.loc[i-4, "TP"] + basetable_K.loc[i-3, "TP"] + basetable_K.loc[i-2,
"TP"]
                        + basetable_K.loc[i-1, "TP"] + basetable_K.loc[i, "TP"]) / 20
    #Create Mean Deviation
    basetable_K.loc[i, "MD"] = (abs(basetable_K.loc[i-19, "TP"] - basetable_K.loc[i, "SMA_TP"])
                        + abs(basetable_K.loc[i-18, "TP"] - basetable_K.loc[i, "SMA_TP"])
                        + abs(basetable_K.loc[i-17, "TP"] - basetable_K.loc[i, "SMA_TP"])
                        + abs(basetable_K.loc[i-16, "TP"] - basetable_K.loc[i, "SMA_TP"])
                        + abs(basetable_K.loc[i-15, "TP"] - basetable_K.loc[i, "SMA_TP"])
                        + abs(basetable_K.loc[i-14, "TP"] - basetable_K.loc[i, "SMA_TP"])
                        + abs(basetable_K.loc[i-13, "TP"] - basetable_K.loc[i, "SMA_TP"])
                        + abs(basetable_K.loc[i-12, "TP"] - basetable_K.loc[i, "SMA_TP"])
                        + abs(basetable_K.loc[i-11, "TP"] - basetable_K.loc[i, "SMA_TP"])
                        + abs(basetable_K.loc[i-10, "TP"] - basetable_K.loc[i, "SMA_TP"])
```

```
                    + abs(basetable_K.loc[i-9, "TP"] - basetable_K.loc[i, "SMA_TP"])
                    + abs(basetable_K.loc[i-8, "TP"] - basetable_K.loc[i, "SMA_TP"])
                    + abs(basetable_K.loc[i-7, "TP"] - basetable_K.loc[i, "SMA_TP"])
                    + abs(basetable_K.loc[i-6, "TP"] - basetable_K.loc[i, "SMA_TP"])
                    + abs(basetable_K.loc[i-5, "TP"] - basetable_K.loc[i, "SMA_TP"])
                    + abs(basetable_K.loc[i-4, "TP"] - basetable_K.loc[i, "SMA_TP"])
                    + abs(basetable_K.loc[i-3, "TP"] - basetable_K.loc[i, "SMA_TP"])
                    + abs(basetable_K.loc[i-2, "TP"] - basetable_K.loc[i, "SMA_TP"])
                    + abs(basetable_K.loc[i-1, "TP"] - basetable_K.loc[i, "SMA_TP"])
                    + abs(basetable_K.loc[i, "TP"] - basetable_K.loc[i, "SMA_TP"]))/20

    #Create the CCI
    basetable_K.loc[i, "CCI"] = ((basetable_K.loc[i, "TP"] - basetable_K.loc[i, "SMA_TP"])
                    / (0.015 * basetable_K.loc[i, "MD"]))


# In[ ]:

basetable_CCI = basetable_K

basetable_CCI = basetable_CCI.drop(basetable_CCI.columns[[13,15, 16]], axis = 1)


# In[ ]:

basetable_CCI.to_csv("basetable_CCI.csv")


# In[ ]:

basetable_CCI["Gain"] = 0
basetable_CCI["Loss"] = 0
```

```
basetable_CCI["avg_Gain"] = 0
basetable_CCI["avg_Loss"] = 0
basetable_CCI["RSI"] = 0
for i in range(1, 25): #we can't start at 0, as we do "i-1"
   if (basetable_CCI.loc[i, "Close"] > basetable_CCI.loc[i-1, "Close"]):
      basetable_CCI.loc[i, "Gain"] = basetable_CCI.loc[i, "Close"] - basetable_CCI.loc[i - 1, "Close"]
      basetable_CCI.loc[i, "Loss"] = 0
   else:
      basetable_CCI.loc[i, "Gain"] = 0
      basetable_CCI.loc[i, "Loss"] = basetable_CCI.loc[i - 1, "Close"] - basetable_CCI.loc[i, "Close"]

for i in range(25, basetable_CCI.shape[0]):
   if (basetable_CCI.loc[i, "Close"] > basetable_CCI.loc[i-1, "Close"]):
      basetable_CCI.loc[i, "Gain"] = basetable_CCI.loc[i, "Close"] - basetable_CCI.loc[i - 1, "Close"]
      basetable_CCI.loc[i, "Loss"] = 0
   else:
      basetable_CCI.loc[i, "Gain"] = 0
      basetable_CCI.loc[i, "Loss"] = basetable_CCI.loc[i - 1, "Close"] - basetable_CCI.loc[i, "Close"]

   basetable_CCI.loc[i, "avg_Gain"] = (basetable_CCI.loc[i-13, "Gain"] + basetable_CCI.loc[i-12,
"Gain"]
                        + basetable_CCI.loc[i-11, "Gain"] + basetable_CCI.loc[i-10, "Gain"]
                        + basetable_CCI.loc[i-9, "Gain"] + basetable_CCI.loc[i-8, "Gain"]
                        + basetable_CCI.loc[i-7, "Gain"] + basetable_CCI.loc[i-6, "Gain"]
                        + basetable_CCI.loc[i-5, "Gain"] + basetable_CCI.loc[i-4, "Gain"]
                        + basetable_CCI.loc[i-3, "Gain"] + basetable_CCI.loc[i-2, "Gain"]
                        + basetable_CCI.loc[i-1, "Gain"] + basetable_CCI.loc[i, "Gain"])/14

   basetable_CCI.loc[i, "avg_Loss"] = (basetable_CCI.loc[i-13, "Loss"] + basetable_CCI.loc[i-12,
"Loss"]
                        + basetable_CCI.loc[i-11, "Loss"] + basetable_CCI.loc[i-10, "Loss"]
                        + basetable_CCI.loc[i-9, "Loss"] + basetable_CCI.loc[i-8, "Loss"]
```

```
                    + basetable_CCI.loc[i-7, "Loss"] + basetable_CCI.loc[i-6, "Loss"]
                    + basetable_CCI.loc[i-5, "Loss"] + basetable_CCI.loc[i-4, "Loss"]
                    + basetable_CCI.loc[i-3, "Loss"] + basetable_CCI.loc[i-2, "Loss"]
                    + basetable_CCI.loc[i-1, "Loss"] + basetable_CCI.loc[i, "Loss"])/14

  basetable_CCI.loc[i, "RSI"] = (100 - 100/(1 + (basetable_CCI.loc[i, "avg_Gain"]/basetable_CCI.loc[i, "avg_Loss"])))


# In[ ]:

#Again it appears we have 1 missing value, because of a division by 0. We will impute it again with the preceding value.
basetable_CCI = basetable_CCI.fillna(method='ffill')
basetable_CCI = basetable_CCI.drop(basetable_CCI.columns[[14, 15, 16, 17]], axis = 1)


# In[ ]:

basetable_RSI = basetable_CCI
basetable_RSI.to_csv("basetable_RSI.csv")


# In[ ]:

#Let's move over to the standard deviation
#for every observation we will calculate the stddev of the past 5, 10 and 20 closing prices.
basetable_RSI["Stddev_5"] = 0
basetable_RSI["Stddev_10"] = 0
basetable_RSI["Stddev_20"] = 0
for i in range(25, basetable_RSI.shape[0]):
```

```
    basetable_RSI.loc[i, "Stddev_5"] = np.std(np.array([basetable_RSI.loc[i-4, "Close"],
basetable_RSI.loc[i-3, "Close"],
                                basetable_RSI.loc[i-2, "Close"], basetable_RSI.loc[i-1, "Close"],
                                  basetable_RSI.loc[i, "Close"]]))

    basetable_RSI.loc[i, "Stddev_10"] = np.std(np.array([basetable_RSI.loc[i-9, "Close"],
basetable_RSI.loc[i-8, "Close"],
                                      basetable_RSI.loc[i-7, "Close"], basetable_RSI.loc[i-6, "Close"],
                                      basetable_RSI.loc[i-5, "Close"],
                                      basetable_RSI.loc[i-4, "Close"], basetable_RSI.loc[i-3, "Close"],
                                      basetable_RSI.loc[i-2, "Close"], basetable_RSI.loc[i-1, "Close"],
                                        basetable_RSI.loc[i, "Close"]]))

    basetable_RSI.loc[i, "Stddev_20"] = np.std(np.array([basetable_RSI.loc[i-19, "Close"],
basetable_RSI.loc[i-18, "Close"],
                                      basetable_RSI.loc[i-17, "Close"], basetable_RSI.loc[i-16, "Close"],
                                      basetable_RSI.loc[i-15, "Close"],
                                      basetable_RSI.loc[i-14, "Close"], basetable_RSI.loc[i-13, "Close"],
                                      basetable_RSI.loc[i-12, "Close"], basetable_RSI.loc[i-11, "Close"],
                                     basetable_RSI.loc[i-10, "Close"],
                                     basetable_RSI.loc[i-9, "Close"], basetable_RSI.loc[i-8, "Close"],
                                      basetable_RSI.loc[i-7, "Close"], basetable_RSI.loc[i-6, "Close"],
                                      basetable_RSI.loc[i-5, "Close"],
                                      basetable_RSI.loc[i-4, "Close"], basetable_RSI.loc[i-3, "Close"],
                                      basetable_RSI.loc[i-2, "Close"], basetable_RSI.loc[i-1, "Close"],
                                     basetable_RSI.loc[i, "Close"]]))


# In[ ]:


basetable_Stddev = basetable_RSI
basetable_Stddev.to_csv("basetable_Stddev.csv")
```

```
basetable_RSI["Upper_band"] = 0
basetable_RSI["Lower_band"] = 0
for i in range(25, basetable_RSI.shape[0]):
    basetable_RSI.loc[i, "Upper_band"] = basetable_RSI.loc[i, "MA_20"] + 2*basetable_RSI.loc[i, "Stddev_20"]
    basetable_RSI.loc[i, "Lower_band"] = basetable_RSI.loc[i, "MA_20"] - 2*basetable_RSI.loc[i, "Stddev_20"]


# In[ ]:

basetable = basetable_RSI

print("bezig met basetable final nu, dit neemt ongeveer 30u in beslag")
# In[ ]:

basetable["Label"] = 0
for i in range(25, basetable.shape[0] - 1):
    if(basetable.loc[i + 1, "Close"] > basetable.loc[i, "Close"]):
        basetable.loc[i, "Label"] = 1
    #print(i)


# In[ ]:

basetable.to_csv("basetable_final_USD_JPY.csv")
```

Appendix D: Second variable creation

```
import pandas as pd
```

```
import numpy as np
import talib

# In[1]:

data = pd.read_csv("E:/Thesis_Simon_Serrarens/Python/basetable_final_USD_JPY.csv")

#When reading in data, a new column with rownumbers is automatically created. we need to drop this
data = data.drop(data.columns[[0]], axis = 1)

#Drop the labels, so that it can be added as the last column later on.
labels = data.iloc[:,20:21]
data = data.drop(data.columns[[20]], axis = 1)
print("labels dropped")
#Test creating additionnal indicators, based on literature

# 1. Create Average directional movement index
data["ADX"] = talib.ADX(high = data.High.values, low = data.Low.values, close = data.Close.values, timeperiod=14)
print (1, "adx created ")

# 2. Aroon
data["AroonDown"], data["AroonUp"] = talib.AROON(high = data.High.values, low = data.Low.values, timeperiod = 14)

print(2, "aroon created")

# 3. Chande momentum oscillator
data["CMO"] = talib.CMO(data.Close.values, timeperiod = 14)
print(3, "Chande created")
```

```
# 4. Directional movement index
data["DX"] = talib.DX(high = data.High.values, low = data.Low.values, close = data.Close.values,
timeperiod = 14)
print(4, "DX created")

# 5. Momentum
data["Momentum"] = talib.MOM(data.Close.values, timeperiod=10)
print(5, "momentum created")

# 6. Percentage price oscillator
data["PPO"] = talib.PPO(data.Close.values, fastperiod=12, slowperiod=26, matype=0)
print(6, "PPO created")

# 7. Rate of change percentage
data["ROCP"] = talib.ROCP(data.Close.values, timeperiod=10)
print(7, "ROCP created")

# 8. Ultimate Oscillator
data["ULTOSC"] = talib.ULTOSC(high = data.High.values, low = data.Low.values, close =
data.Close.values, timeperiod1 = 7, timeperiod2 = 14, timeperiod3 = 28)

# 9. DEMA
data["DEMA5"] = talib.DEMA(data.Close.values, timeperiod=5)
data["DEMA10"] = talib.DEMA(data.Close.values, timeperiod=10)
data["DEMA20"] = talib.DEMA(data.Close.values, timeperiod=20)
print(9, "Dema created")

# 10. WMA
data["WMA5"] = talib.WMA(data.Close.values, timeperiod=5)
data["WMA10"] = talib.WMA(data.Close.values, timeperiod=10)
data["WMA20"] = talib.WMA(data.Close.values, timeperiod=20)
```

```
# 11. average true range
data["ATR"] = talib.ATR(high = data.High.values, low = data.Low.values, close = data.Close.values,
timeperiod=14)
print(11, "atr created")

data["Labels"] = labels

data.to_csv("E:/Thesis_Simon_Serrarens/Data/basetable_final_USD_JPY_v2.csv")
```

Appendix E: Data normalisation

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler


# In[36]:

data = pd.read_csv("basetable_final_USD_JPY_v2.csv")
data = data.drop(data.columns[[0]], axis = 1)
pd.options.mode.use_inf_as_na = True

data = data.fillna(method="ffill")


#volledige set

features = data.iloc[:,0:36]
labels = data.iloc[:,36:37]

#We will first split in 70/30, for train and rest, and then split the rest in validation and test.
#X_train, X_rest, y_train, y_rest = train_test_split(features, labels, shuffle=False, train_size=0.7,
test_size=0.3)
```

```
#X_val, X_test, y_val, y_test = train_test_split(X_rest, y_rest, shuffle=False, train_size=0.5, test_size=0.5)


#Full split
#we laten de eerste 39 vallen, die zijn gebruikt geweest om variabelen mee aan te maken.
X_train = features[39:1493180]
X_val = features[1493180:1866475]
X_test = features[1866475:]

y_train = labels[39:1493180]
y_val = labels[1493180:1866475]
y_test = labels[1866475:]

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)

#Now we reassemble to be able to make our timewindow segments
features = pd.concat([pd.DataFrame(X_train), pd.DataFrame(X_val), pd.DataFrame(X_test)])
labels = pd.concat([pd.DataFrame(y_train), pd.DataFrame(y_val), pd.DataFrame(y_test)])

#We write this back to csv, to be able to then make the windows.
#first reattach the labels
features["Labels"] = labels

features.to_csv("basetable_normalized_USD_JPY.csv")
```

Appendix F: Data windowing

```
import pandas as pd
import numpy as np
```

```
data = pd.read_csv("basetable_normalized_USD_JPY.csv")
data = data.drop(data.columns[[0]], axis = 1)

#deel 1
features = data.iloc[0:25000,0:36]
labels = data.iloc[0:25000,36:37]

labels = labels.fillna(0)
def windows(nrows, size):
    start, step = 0, 1
    while start < (nrows - size):
        yield start, start + size
        start += step

#This is the code to create a 3D matrix that will hold the input. One dimension will be the input features, another
#is the number of timesteps that one input will contain, and the third dimension is the total number of observations in
#the data

def segment_data(features, labels, window_size = 40):
    #Important, note that we initialize here two empty arrays! segments and segment_labels,
    #But these "empty" arrays actually contain a first element, which is all zeroes. We need to deleted these after
    #segmenting the data

    #For the labels this doesn't seem to occur, so we won't take out the first element there.
    segments = np.empty((window_size,36))
    segment_labels = np.empty((0))
    nrows = len(features)

    for (start, end) in windows(nrows, window_size):
```

```
        if (start % 1000 == 0):
            print(start)
        if (len(data.iloc[start:end]) == window_size):
            segment = features[start:end]
            label = labels.iloc[end-1,]
            #print(label)
            segments = np.vstack([segments, segment])
            segment_labels = np.append(segment_labels, label)
            #print(start)
    segments = segments.reshape(-1, window_size,36) #Since we have 20 features
    segment_labels = segment_labels.reshape(-1)
    return segments, segment_labels

segments, labels = segment_data(features, labels)
segments = segments[1:]

#Important! Change these to the right folder!
np.save("E:/Thesis_Simon_Serrarens/Python/labels1.npy", labels)
np.save("E:/Thesis_Simon_Serrarens/Python/segments1.npy", segments)

features = data.iloc[24960:50000,0:36] #We will drop the first 25 rows, as these are incomplete
labels = data.iloc[24960:50000,36:37]

labels = labels.fillna(0)

def windows(nrows, size):
    start, step = 0, 1
    while start < (nrows - size):
        yield start, start + size
        start += step
def segment_data(features, labels, window_size = 40):
    #Important, note that we initialize here two empty arrays! segments and segment_labels,
```

```
    #But these "empty" arrays actually contain a first element, which is all zeroes. We need to deleted these after
    #segmenting the data

    #For the labels this doesn't seem to occur, so we won't take out the first element there.
    segments = np.empty((window_size,36))
    segment_labels = np.empty((0))
    nrows = len(features)

    for (start, end) in windows(nrows, window_size):
        if (start % 1000 == 0):
            print(start)
        if (len(data.iloc[start:end]) == window_size):
            segment = features[start:end]
            label = labels.iloc[end-1,]
            #print(label)
            segments = np.vstack([segments, segment])
            segment_labels = np.append(segment_labels, label)
            #print(start)
    segments = segments.reshape(-1, window_size,36) #Since we have 20 features
    segment_labels = segment_labels.reshape(-1)
    return segments, segment_labels


segments, labels = segment_data(features, labels)


segments = segments[1:]
#Important! Change these to the right folder!
np.save("E:/Thesis_Simon_Serrarens/Python/labels2.npy", labels)
np.save("E:/Thesis_Simon_Serrarens/Python/segments2.npy", segments)
```

```
features = data.iloc[49960:75000,0:36] #We will drop the first 25 rows, as these are incomplete
labels = data.iloc[49960:75000,36:37]

labels = labels.fillna(0)

def windows(nrows, size):
    start, step = 0, 1
    while start < (nrows - size):
        yield start, start + size
        start += step
def segment_data(features, labels, window_size = 40):
    #Important, note that we initialize here two empty arrays! segments and segment_labels,
    #But these "empty" arrays actually contain a first element, which is all zeroes. We need to deleted these after
    #segmenting the data

    #For the labels this doesn't seem to occur, so we won't take out the first element there.
    segments = np.empty((window_size,36))
    segment_labels = np.empty((0))
    nrows = len(features)

    for (start, end) in windows(nrows, window_size):
        if (start % 1000 == 0):
            print(start)
        if (len(data.iloc[start:end]) == window_size):
            segment = features[start:end]
            label = labels.iloc[end-1,]
            #print(label)
            segments = np.vstack([segments, segment])
            segment_labels = np.append(segment_labels, label)
            #print(start)
    segments = segments.reshape(-1, window_size,36) #Since we have 20 features
```

```
    segment_labels = segment_labels.reshape(-1)
    return segments, segment_labels

segments, labels = segment_data(features, labels)
segments = segments[1:]

#Important! Change these to the right folder!
np.save("E:/Thesis_Simon_Serrarens/Python/labels3.npy", labels)
np.save("E:/Thesis_Simon_Serrarens/Python/segments3.npy", segments)

features = data.iloc[74960:100000,0:36]
labels = data.iloc[74960:100000,36:37]

labels = labels.fillna(0)

def windows(nrows, size):
    start, step = 0, 1
    while start < (nrows - size):
        yield start, start + size
        start += step


def segment_data(features, labels, window_size = 40):
    #Important, note that we initialize here two empty arrays! segments and segment_labels,
    #But these "empty" arrays actually contain a first element, which is all zeroes. We need to deleted these after
    #segmenting the data

    #For the labels this doesn't seem to occur, so we won't take out the first element there.
    segments = np.empty((window_size,36))
    segment_labels = np.empty((0))
    nrows = len(features)
```

```
    for (start, end) in windows(nrows, window_size):
        if (start % 1000 == 0):
            print(start)
        if (len(data.iloc[start:end]) == window_size):
            segment = features[start:end]
            label = labels.iloc[end-1,]
            #print(label)
            segments = np.vstack([segments, segment])
            segment_labels = np.append(segment_labels, label)
            #print(start)
    segments = segments.reshape(-1, window_size,36) #Since we have 20 features
    segment_labels = segment_labels.reshape(-1)
    return segments, segment_labels

segments, labels = segment_data(features, labels)


segments = segments[1:]
#Important! Change these to the right folder!
np.save("E:/Thesis_Simon_Serrarens/Python/labels4.npy", labels)
np.save("E:/Thesis_Simon_Serrarens/Python/segments4.npy", segments)
features = data.iloc[99960:125000,0:36]
labels = data.iloc[99960:125000,36:37]

labels = labels.fillna(0) #unsolvable at the moment
def windows(nrows, size):
    start, step = 0, 1
    while start < (nrows - size):
        yield start, start + size
        start += step
```

```
def segment_data(features, labels, window_size = 40):
    #Important, note that we initialize here two empty arrays! segments and segment_labels,
    #But these "empty" arrays actually contain a first element, which is all zeroes. We need to deleted these after
    #segmenting the data

    #For the labels this doesn't seem to occur, so we won't take out the first element there.
    segments = np.empty((window_size,36))
    segment_labels = np.empty((0))
    nrows = len(features)

    for (start, end) in windows(nrows, window_size):
        if (start % 1000 == 0):
            print(start)
        if (len(data.iloc[start:end]) == window_size):
            segment = features[start:end]
            label = labels.iloc[end-1,]
            #print(label)
            segments = np.vstack([segments, segment])
            segment_labels = np.append(segment_labels, label)
            #print(start)
    segments = segments.reshape(-1, window_size,36) #Since we have 20 features
    segment_labels = segment_labels.reshape(-1)
    return segments, segment_labels

segments, labels = segment_data(features, labels)

segments = segments[1:]
#Important! Change these to the right folder!
np.save("E:/Thesis_Simon_Serrarens/Python/labels5.npy", labels)
np.save("E:/Thesis_Simon_Serrarens/Python/segments5.npy", segments)
```

```
#deel 1
features = data.iloc[124960:150000,0:36] #We will drop the first 25 rows, as these are incomplete
labels = data.iloc[124960:150000,36:37]

labels = labels.fillna(0)

def windows(nrows, size):
    start, step = 0, 1
    while start < (nrows - size):
        yield start, start + size
        start += step

def segment_data(features, labels, window_size = 40):
    #Important, note that we initialize here two empty arrays! segments and segment_labels,
    #But these "empty" arrays actually contain a first element, which is all zeroes. We need to deleted these after
    #segmenting the data

    #For the labels this doesn't seem to occur, so we won't take out the first element there.
    segments = np.empty((window_size,36))
    segment_labels = np.empty((0))
    nrows = len(features)

    for (start, end) in windows(nrows, window_size):
        if (start % 1000 == 0):
            print(start)
        if (len(data.iloc[start:end]) == window_size):
            segment = features[start:end]
            label = labels.iloc[end-1,]
            #print(label)
            segments = np.vstack([segments, segment])
            segment_labels = np.append(segment_labels, label)
```

```
            #print(start)
    segments = segments.reshape(-1, window_size,36) #Since we have 20 features
    segment_labels = segment_labels.reshape(-1)
    return segments, segment_labels


segments, labels = segment_data(features, labels)
segments = segments[1:]
#Important! Change these to the right folder!
np.save("E:/Thesis_Simon_Serrarens/Python/labels6.npy", labels)
np.save("E:/Thesis_Simon_Serrarens/Python/segments6.npy", segments)


features = data.iloc[149960:175000,0:36] #We will drop the first 25 rows, as these are incomplete
labels = data.iloc[149960:175000,36:37]

labels = labels.fillna(0)

def windows(nrows, size):
    start, step = 0, 1
    while start < (nrows - size):
        yield start, start + size
        start += step

def segment_data(features, labels, window_size = 40):
    #Important, note that we initialize here two empty arrays! segments and segment_labels,
    #But these "empty" arrays actually contain a first element, which is all zeroes. We need to
deleted these after
    #segmenting the data

    #For the labels this doesn't seem to occur, so we won't take out the first element there.
    segments = np.empty((window_size,36))
```

```
    segment_labels = np.empty((0))
    nrows = len(features)

    for (start, end) in windows(nrows, window_size):
        if (start % 1000 == 0):
            print(start)
        if (len(data.iloc[start:end]) == window_size):
            segment = features[start:end]
            label = labels.iloc[end-1,]
            #print(label)
            segments = np.vstack([segments, segment])
            segment_labels = np.append(segment_labels, label)
            #print(start)
    segments = segments.reshape(-1, window_size,36) #Since we have 20 features
    segment_labels = segment_labels.reshape(-1)
    return segments, segment_labels

segments, labels = segment_data(features, labels)



segments = segments[1:]

#Important! Change these to the right folder!
np.save("E:/Thesis_Simon_Serrarens/Python/labels7.npy", labels)
np.save("E:/Thesis_Simon_Serrarens/Python/segments7.npy", segments)



#deel 1
features = data.iloc[174960:200000,0:36]
labels = data.iloc[174960:200000,36:37]

labels = labels.fillna(0)
```

```
def windows(nrows, size):
    start, step = 0, 1
    while start < (nrows - size):
        yield start, start + size
        start += step

def segment_data(features, labels, window_size = 40):
    #Important, note that we initialize here two empty arrays! segments and segment_labels,
    #But these "empty" arrays actually contain a first element, which is all zeroes. We need to deleted these after
    #segmenting the data

    #For the labels this doesn't seem to occur, so we won't take out the first element there.
    segments = np.empty((window_size,36))
    segment_labels = np.empty((0))
    nrows = len(features)

    for (start, end) in windows(nrows, window_size):
        if (start % 1000 == 0):
            print(start)
        if (len(data.iloc[start:end]) == window_size):
            segment = features[start:end]
            label = labels.iloc[end-1,]
            #print(label)
            segments = np.vstack([segments, segment])
            segment_labels = np.append(segment_labels, label)
            #print(start)
    segments = segments.reshape(-1, window_size,36) #Since we have 20 features
    segment_labels = segment_labels.reshape(-1)
    return segments, segment_labels
```

```
segments, labels = segment_data(features, labels)


segments = segments[1:]
#Important! Change these to the right folder!
np.save("E:/Thesis_Simon_Serrarens/Python/labels8.npy", labels)
np.save("E:/Thesis_Simon_Serrarens/Python/segments8.npy", segments)


#deel 1
features = data.iloc[199960:225000,0:36]
labels = data.iloc[199960:225000,36:37]

labels = labels.fillna(0)

def windows(nrows, size):
    start, step = 0, 1
    while start < (nrows - size):
        yield start, start + size
        start += step


def segment_data(features, labels, window_size = 40):
    #Important, note that we initialize here two empty arrays! segments and segment_labels,
    #But these "empty" arrays actually contain a first element, which is all zeroes. We need to deleted these after
    #segmenting the data

    #For the labels this doesn't seem to occur, so we won't take out the first element there.
    segments = np.empty((window_size,36))
    segment_labels = np.empty((0))
    nrows = len(features)
```

```
    for (start, end) in windows(nrows, window_size):
        if (start % 1000 == 0):
            print(start)
        if (len(data.iloc[start:end]) == window_size):
            segment = features[start:end]
            label = labels.iloc[end-1,]
            #print(label)
            segments = np.vstack([segments, segment])
            segment_labels = np.append(segment_labels, label)
            #print(start)
    segments = segments.reshape(-1, window_size,36) #Since we have 20 features
    segment_labels = segment_labels.reshape(-1)
    return segments, segment_labels
segments, labels = segment_data(features, labels)



segments = segments[1:]

#Important! Change these to the right folder!
np.save("E:/Thesis_Simon_Serrarens/Python/labels9.npy", labels)
np.save("E:/Thesis_Simon_Serrarens/Python/segments9.npy", segments)



#deel 1
features = data.iloc[224960:250000,0:36] #We will drop the first 25 rows, as these are incomplete
labels = data.iloc[224960:250000,36:37]

labels = labels.fillna(0)

def windows(nrows, size):
    start, step = 0, 1
```

```
    while start < (nrows - size):
        yield start, start + size
        start += step

def segment_data(features, labels, window_size = 40):
    #Important, note that we initialize here two empty arrays! segments and segment_labels,
    #But these "empty" arrays actually contain a first element, which is all zeroes. We need to
deleted these after
    #segmenting the data

    #For the labels this doesn't seem to occur, so we won't take out the first element there.
    segments = np.empty((window_size,36))
    segment_labels = np.empty((0))
    nrows = len(features)

    for (start, end) in windows(nrows, window_size):
        if (start % 1000 == 0):
            print(start)
        if (len(data.iloc[start:end]) == window_size):
            segment = features[start:end]
            label = labels.iloc[end-1,]
            #print(label)
            segments = np.vstack([segments, segment])
            segment_labels = np.append(segment_labels, label)
            #print(start)
    segments = segments.reshape(-1, window_size,36) #Since we have 20 features
    segment_labels = segment_labels.reshape(-1)
    return segments, segment_labels

segments, labels = segment_data(features, labels)
segments = segments[1:]
#Important! Change these to the right folder!
```

```
np.save("E:/Thesis_Simon_Serrarens/Python/labels10.npy", labels)
np.save("E:/Thesis_Simon_Serrarens/Python/segments10.npy", segments)



#deel 1
features = data.iloc[249960:275000,0:36] #We will drop the first 25 rows, as these are incomplete
labels = data.iloc[249960:275000,36:37]

labels = labels.fillna(0)

def windows(nrows, size):
    start, step = 0, 1
    while start < (nrows - size):
        yield start, start + size
        start += step

# In[46]:

def segment_data(features, labels, window_size = 40):
    #Important, note that we initialize here two empty arrays! segments and segment_labels,
    #But these "empty" arrays actually contain a first element, which is all zeroes. We need to
deleted these after
    #segmenting the data

    #For the labels this doesn't seem to occur, so we won't take out the first element there.
    segments = np.empty((window_size,36))
    segment_labels = np.empty((0))
    nrows = len(features)

    for (start, end) in windows(nrows, window_size):
        if (start % 1000 == 0):
            print(start)
```

```
        if (len(data.iloc[start:end]) == window_size):
            segment = features[start:end]
            label = labels.iloc[end-1,]
            #print(label)
            segments = np.vstack([segments, segment])
            segment_labels = np.append(segment_labels, label)
            #print(start)
    segments = segments.reshape(-1, window_size,36) #Since we have 20 features
    segment_labels = segment_labels.reshape(-1)
    return segments, segment_labels



segments, labels = segment_data(features, labels)
segments = segments[1:]

#Important! Change these to the right folder!
np.save("E:/Thesis_Simon_Serrarens/Python/labels11.npy", labels)
np.save("E:/Thesis_Simon_Serrarens/Python/segments11.npy", segments)

features = data.iloc[274960:300000,0:36]
labels = data.iloc[274960:300000,36:37]

labels = labels.fillna(0) #unsolvable at the moment

def windows(nrows, size):
    start, step = 0, 1
    while start < (nrows - size):
        yield start, start + size
        start += step
def segment_data(features, labels, window_size = 40):
    #Important, note that we initialize here two empty arrays! segments and segment_labels,
```

```
    #But these "empty" arrays actually contain a first element, which is all zeroes. We need to deleted these after
    #segmenting the data

    #For the labels this doesn't seem to occur, so we won't take out the first element there.
    segments = np.empty((window_size,36))
    segment_labels = np.empty((0))
    nrows = len(features)

    for (start, end) in windows(nrows, window_size):
        if (start % 1000 == 0):
            print(start)
        if (len(data.iloc[start:end]) == window_size):
            segment = features[start:end]
            label = labels.iloc[end-1,]
            #print(label)
            segments = np.vstack([segments, segment])
            segment_labels = np.append(segment_labels, label)
            #print(start)
    segments = segments.reshape(-1, window_size,36) #Since we have 20 features
    segment_labels = segment_labels.reshape(-1)
    return segments, segment_labels

segments, labels = segment_data(features, labels)
segments = segments[1:]

#Important! Change these to the right folder!
np.save("E:/Thesis_Simon_Serrarens/Python/labels12.npy", labels)
np.save("E:/Thesis_Simon_Serrarens/Python/segments12.npy", segments)
```

Appendix G: The MLP model

```
#Necessary imports
import tensorflow as tf
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import f1_score
import time


# In[ ]:


#Load in all the segments
segments1 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments1.npy")
segments2 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments2.npy")
segments3 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments3.npy")
segments4 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments4.npy")
segments5 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments5.npy")
segments6 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments6.npy")
segments7 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments7.npy")
segments8 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments8.npy")
segments9 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments9.npy")
segments10 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments10.npy")
segments11 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments11.npy")
segments12 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments12.npy")
segments13 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments13.npy")
segments14 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments14.npy")
segments15 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments15.npy")
segments16 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments16.npy")
segments17 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments17.npy")
segments18 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments18.npy")
segments19 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments19.npy")
```

```
segments20 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments20.npy")
segments21 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments21.npy")
segments22 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments22.npy")
segments23 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments23.npy")
segments24 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments24.npy")
segments25 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments25.npy")
segments26 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments26.npy")
segments27 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments27.npy")
segments28 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments28.npy")
segments29 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments29.npy")
segments30 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments30.npy")
segments31 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments31.npy")
segments32 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments32.npy")
segments33 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments33.npy")
segments34 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments34.npy")
segments35 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments35.npy")
segments36 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments36.npy")
segments37 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments37.npy")
segments38 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments38.npy")
segments39 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments39.npy")
segments40 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments40.npy")
segments41 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments41.npy")
segments42 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments42.npy")
segments43 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments43.npy")
segments44 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments44.npy")
segments45 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments45.npy")
segments46 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments46.npy")
segments47 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments47.npy")
segments48 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments48.npy")
segments49 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments49.npy")
segments50 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments50.npy")
segments51 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments51.npy")
```

```
segments52 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments52.npy")
segments53 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments53.npy")
segments54 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments54.npy")
segments55 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments55.npy")
segments56 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments56.npy")
segments57 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments57.npy")
segments58 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments58.npy")
segments59 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments59.npy")
segments60 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments60.npy")
segments61 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments61.npy")
segments62 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments62.npy")
segments63 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments63.npy")
segments64 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments64.npy")
segments65 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments65.npy")
segments66 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments66.npy")
segments67 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments67.npy")
segments68 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments68.npy")
segments69 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments69.npy")
segments70 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments70.npy")
segments71 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments71.npy")
segments72 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments72.npy")
segments73 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments73.npy")
segments74 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments74.npy")
segments75 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments75.npy")
segments76 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments76.npy")
segments77 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments77.npy")
segments78 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments78.npy")
segments79 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments79.npy")
segments80 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments80.npy")
segments81 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments81.npy")
segments82 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments82.npy")
segments83 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments83.npy")
```

```
segments84 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments84.npy")
segments85 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments85.npy")
segments86 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments86.npy")
segments87 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments87.npy")
segments88 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments88.npy")
segments89 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments89.npy")
segments90 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/segments90.npy")

#Load in all the labels
labels1 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels1.npy")
labels2 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels2.npy")
labels3 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels3.npy")
labels4 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels4.npy")
labels5 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels5.npy")
labels6 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels6.npy")
labels7 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels7.npy")
labels8 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels8.npy")
labels9 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels9.npy")
labels10 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels10.npy")
labels11 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels11.npy")
labels12 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels12.npy")
labels13 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels13.npy")
labels14 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels14.npy")
labels15 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels15.npy")
labels16 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels16.npy")
labels17 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels17.npy")
labels18 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels18.npy")
labels19 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels19.npy")
labels20 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels20.npy")
labels21 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels21.npy")
labels22 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels22.npy")
labels23 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels23.npy")
```

```
labels24 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels24.npy")
labels25 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels25.npy")
labels26 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels26.npy")
labels27 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels27.npy")
labels28 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels28.npy")
labels29 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels29.npy")
labels30 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels30.npy")
labels31 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels31.npy")
labels32 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels32.npy")
labels33 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels33.npy")
labels34 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels34.npy")
labels35 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels35.npy")
labels36 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels36.npy")
labels37 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels37.npy")
labels38 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels38.npy")
labels39 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels39.npy")
labels40 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels40.npy")
labels41 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels41.npy")
labels42 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels42.npy")
labels43 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels43.npy")
labels44 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels44.npy")
labels45 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels45.npy")
labels46 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels46.npy")
labels47 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels47.npy")
labels48 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels48.npy")
labels49 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels49.npy")
labels50 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels50.npy")
labels51 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels51.npy")
labels52 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels52.npy")
labels53 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels53.npy")
labels54 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels54.npy")
labels55 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels55.npy")
```

```
labels56 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels56.npy")
labels57 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels57.npy")
labels58 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels58.npy")
labels59 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels59.npy")
labels60 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels60.npy")
labels61 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels61.npy")
labels62 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels62.npy")
labels63 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels63.npy")
labels64 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels64.npy")
labels65 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels65.npy")
labels66 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels66.npy")
labels67 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels67.npy")
labels68 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels68.npy")
labels69 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels69.npy")
labels70 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels70.npy")
labels71 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels71.npy")
labels72 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels72.npy")
labels73 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels73.npy")
labels74 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels74.npy")
labels75 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels75.npy")
labels76 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels76.npy")
labels77 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels77.npy")
labels78 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels78.npy")
labels79 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels79.npy")
labels80 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels80.npy")
labels81 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels81.npy")
labels82 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels82.npy")
labels83 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels83.npy")
labels84 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels84.npy")
labels85 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels85.npy")
labels86 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels86.npy")
labels87 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels87.npy")
```

```
labels88 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels88.npy")
labels89 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels89.npy")
labels90 = np.load("E:/Thesis_Simon_Serrarens/Python/segments_labels/labels90.npy")

print("all files are read in.")
# In[ ]:

#Combine all segments again
segments = np.vstack([segments1, segments2, segments3, segments4, segments5, segments6,
segments7, segments8,
            segments9, segments10, segments11, segments12, segments13, segments14,
segments15, segments16,
            segments17, segments18, segments19, segments20, segments21, segments22,
                                    segments23, segments24, segments25, segments26,
segments27, segments28,
                                    segments29, segments30, segments31, segments32,
segments33, segments34,
                                    segments35 ,segments36, segments37, segments38,
segments39, segments40,
                                    segments41, segments42, segments43, segments44,
segments45, segments46,
                                    segments47, segments48, segments49, segments50,
segments51,segments52,
                                    segments53, segments54, segments55, segments56,
segments57, segments58,
                                    segments59, segments60, segments61, segments62,
segments63, segments64,
                                    segments65, segments66, segments67, segments68,
segments69, segments70,
                                    segments71, segments72, segments73, segments74,
segments75, segments76,
```

```
                                    segments77, segments78, segments79, segments80,
segments81, segments82,
                                    segments83, segments84, segments85, segments86,
segments87, segments88,
                                    segments89, segments90
                                    ])
#combine all labels again
labels = np.append(labels1, labels2)
labels = np.append(labels, labels3)
labels = np.append(labels, labels4)
labels = np.append(labels, labels5)
labels = np.append(labels, labels6)
labels = np.append(labels, labels7)
labels = np.append(labels, labels8)
labels = np.append(labels, labels9)
labels = np.append(labels, labels10)
labels = np.append(labels, labels11)
labels = np.append(labels, labels12)
labels = np.append(labels, labels13)
labels = np.append(labels, labels14)
labels = np.append(labels, labels15)
labels = np.append(labels, labels16)
labels = np.append(labels, labels17)
labels = np.append(labels, labels18)
labels = np.append(labels, labels19)
labels = np.append(labels, labels20)
labels = np.append(labels, labels21)
labels = np.append(labels, labels22)
labels = np.append(labels, labels23)
labels = np.append(labels, labels24)
labels = np.append(labels, labels25)
labels = np.append(labels, labels26)
```

```
labels = np.append(labels, labels27)
labels = np.append(labels, labels28)
labels = np.append(labels, labels29)
labels = np.append(labels, labels30)
labels = np.append(labels, labels31)
labels = np.append(labels, labels32)
labels = np.append(labels, labels33)
labels = np.append(labels, labels34)
labels = np.append(labels, labels35)
labels = np.append(labels, labels36)
labels = np.append(labels, labels37)
labels = np.append(labels, labels38)
labels = np.append(labels, labels39)
labels = np.append(labels, labels40)
labels = np.append(labels, labels41)
labels = np.append(labels, labels42)
labels = np.append(labels, labels43)
labels = np.append(labels, labels44)
labels = np.append(labels, labels45)
labels = np.append(labels, labels46)
labels = np.append(labels, labels47)
labels = np.append(labels, labels48)
labels = np.append(labels, labels49)
labels = np.append(labels, labels50)
labels = np.append(labels, labels51)
labels = np.append(labels, labels52)
labels = np.append(labels, labels53)
labels = np.append(labels, labels54)
labels = np.append(labels, labels55)
labels = np.append(labels, labels56)
labels = np.append(labels, labels57)
labels = np.append(labels, labels58)
```

```
labels = np.append(labels, labels59)
labels = np.append(labels, labels60)
labels = np.append(labels, labels61)
labels = np.append(labels, labels62)
labels = np.append(labels, labels63)
labels = np.append(labels, labels64)
labels = np.append(labels, labels65)
labels = np.append(labels, labels66)
labels = np.append(labels, labels67)
labels = np.append(labels, labels68)
labels = np.append(labels, labels69)
labels = np.append(labels, labels70)
labels = np.append(labels, labels71)
labels = np.append(labels, labels72)
labels = np.append(labels, labels73)
labels = np.append(labels, labels74)
labels = np.append(labels, labels75)
labels = np.append(labels, labels76)
labels = np.append(labels, labels77)
labels = np.append(labels, labels78)
labels = np.append(labels, labels79)
labels = np.append(labels, labels80)
labels = np.append(labels, labels81)
labels = np.append(labels, labels82)
labels = np.append(labels, labels83)
labels = np.append(labels, labels84)
labels = np.append(labels, labels85)
labels = np.append(labels, labels86)
labels = np.append(labels, labels87)
labels = np.append(labels, labels88)
labels = np.append(labels, labels89)
labels = np.append(labels, labels90)
```

```
print("data fully prepared")

# In[ ]:

#Split again in train, test and validation sets

X_train = segments[0:1493180]
X_val = segments[1493180:1866475]
X_test = segments[1866475:]

y_train = labels[0:1493180]
y_val = labels[1493180:1866475]
y_test = labels[1866475:]



#Construction
tf.reset_default_graph()
n_steps = 40
n_inputs = 36
n_neurons = 36
n_hidden1 = 36
n_hidden2 = 36
n_outputs = 2
n_layers = 2
learning_rate = 0.01
batch_size = 5000
total_batches = X_train.shape[0] // batch_size

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
X_reshaped = tf.reshape(X, [-1, n_steps*n_inputs])
y = tf.placeholder(tf.int32, shape = (None))
```

```
keep_prob = tf.placeholder_with_default(1.0, shape=())

training = tf.placeholder_with_default(False, shape=(), name='training')

dropout_rate = 0.5  # == 1 - keep_prob
X_drop = tf.layers.dropout(X_reshaped, dropout_rate, training=training)

hidden1 = tf.layers.dense(X_drop, n_hidden1, activation=tf.nn.relu,
                          name="hidden1")
hidden1_drop = tf.layers.dropout(hidden1, dropout_rate, training=training)
hidden2 = tf.layers.dense(hidden1_drop, n_hidden2, activation=tf.nn.relu,
                          name="hidden2")
hidden2_drop = tf.layers.dropout(hidden2, dropout_rate, training=training)

logits = tf.layers.dense(hidden2_drop, n_outputs)

xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
loss = tf.reduce_mean(xentropy, name="loss")

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

correct = tf.nn.in_top_k(logits, y, 1)
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

prediction = tf.argmax(logits, 1)

init = tf.group(tf.global_variables_initializer(), tf.local_variables_initializer())
saver = tf.train.Saver()

session_conf = tf.ConfigProto(
```

```
intra_op_parallelism_threads=12,
inter_op_parallelism_threads=12)


# In[ ]:

#Benchmark with 100 epochs, full data
n_epochs = 100
max_worse_iter = 10
best_acc_val = -9999

start = time.time()
print("Training started")
with tf.Session(config = session_conf) as sess:
    init.run()
    for epoch in range(n_epochs):
        count = 0
        for b in range(X_train.shape[0] // batch_size):
            offset = (b * batch_size)# % (X_train.shape[0] - batch_size)
            X_batch = X_train[offset:(offset + batch_size), : , :]
            y_batch = y_train[offset:(offset + batch_size)]
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch,
                                     training: True})
            count += 1
            if (b % 40 == 0):
                acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})


                print (b*batch_size, "Train accuracy = ", acc_train)


            if (count % 160 == 0):
```

```
            print("Midpoint reached, halfway epoch, calculating validation accuracy...")
            acc_val = accuracy.eval(feed_dict={X: X_val, y: y_val})

            print("Validation accuracy at half epoch:", acc_val)

            if (acc_val > best_acc_val):
                print("Best validation accuracy so far!")
                best_acc_val = acc_val
                iterations_since_best_acc = 0

                save_path = saver.save(sess,
"E:/Thesis_Simon_Serrarens/Python/tmp/testmodel_bench_simple_mlp.ckpt")
            else:
                print("No improvement at this point, continue learning till full epoch.")




        print("End of epoch, calculating validation accuracy...")
        acc_val = accuracy.eval(feed_dict={X: X_val, y: y_val})

        print("Epoch", epoch, "Train accuracy:", acc_train, "Validation accuracy", acc_val)



        if (acc_val > best_acc_val):
            print("Best validation accuracy so far!")
            best_acc_val = acc_val
            iterations_since_best_acc = 0

            save_path = saver.save(sess,
"E:/Thesis_Simon_Serrarens/Python/tmp/testmodel_bench_simple_mlp.ckpt")
```

```
        else:
            iterations_since_best_acc += 1
            print("Epochs since last best validation accuracy:", iterations_since_best_acc)
            if iterations_since_best_acc == max_worse_iter:
                print("Early stopping")
                end = time.time()
                tot_time = end - start
                print("The training time for", epoch, " epochs is about:", tot_time)
                break



with tf.Session(config= session_conf) as sess:
    init.run()
    saver.restore(sess,
"E:/Thesis_Simon_Serrarens/Python/tmp/testmodel_bench_simple_mlp.ckpt")
    acc_test = 0
    n_batches = 0



    acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})


    print("The test accuracy of this model is ", acc_test)
```

Appendix H: The LSTM model

Important: as in the MLP model, this script should read in the segments and labels first. This was cut out here to save space.

```
#Split again in train, test and validation sets

X_train = segments[0:1493180]
X_val = segments[1493180:1866475]
X_test = segments[1866475:]
```

```
y_train = labels[0:1493180]
y_val = labels[1493180:1866475]
y_test = labels[1866475:]


#Construction
tf.reset_default_graph()
n_steps = 40
n_inputs = 36
n_neurons = 36
n_outputs = 2
n_layers = 2
learning_rate = 0.01
batch_size = 5000
total_batches = X_train.shape[0] // batch_size

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.int32, [None])

keep_prob = tf.placeholder_with_default(1.0, shape=())


lstm_cells = [tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons, activation = tf.nn.relu)
        for layer in range(n_layers)]
cells_drop = [tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob = keep_prob)
for cell in lstm_cells]
multi_cell = tf.contrib.rnn.MultiRNNCell(cells_drop)
outputs, states = tf.nn.dynamic_rnn(multi_cell, X, dtype=tf.float32)

top_layer_h_state = states[-1][1]
logits = tf.layers.dense(top_layer_h_state, n_outputs, name="softmax")
```

```
xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
loss = tf.reduce_mean(xentropy, name="loss")

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)



correct = tf.nn.in_top_k(logits, y, 1)
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.group(tf.global_variables_initializer(), tf.local_variables_initializer())
saver = tf.train.Saver()



session_conf = tf.ConfigProto(
intra_op_parallelism_threads=12,
inter_op_parallelism_threads=12)



# In[ ]:

#Benchmark with 100 epochs, full data
n_epochs = 100
train_keep_prob = 0.5
max_worse_iter = 10
best_acc_val = -9999

start = time.time()
print("Training started")
with tf.Session(config = session_conf) as sess:
    init.run()
```

```
    for epoch in range(n_epochs):
        count = 0
        for b in range(X_train.shape[0] // batch_size):
            offset = (b * batch_size)# % (X_train.shape[0] - batch_size)
            X_batch = X_train[offset:(offset + batch_size), : , :]
            y_batch = y_train[offset:(offset + batch_size)]
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch,
                                     keep_prob: train_keep_prob})
            count += 1
            if (b % 40 == 0):
                acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})

                print (b*batch_size, "Train accuracy = ", acc_train)

            if (count % 160 == 0):
                print("Midpoint reached, halfway epoch, calculating validation accuracy...")
                acc_val = accuracy.eval(feed_dict={X: X_val, y: y_val})

                print("Validation accuracy at half epoch:", acc_val)

                if (acc_val > best_acc_val):
                    print("Best validation accuracy so far!")
                    best_acc_val = acc_val
                    iterations_since_best_acc = 0

                    save_path = saver.save(sess,
"E:/Thesis_Simon_Serrarens/Python/tmp/testmodel_bench_simple.ckpt")
                else:
                    print("No improvement at this point, continue learning till full epoch.")
```

```
        print("End of epoch, calculating validation accuracy and AUC...")
        acc_val = accuracy.eval(feed_dict={X: X_val, y: y_val})

        print("Epoch", epoch, "Train accuracy:", acc_train, "Validation accuracy", acc_val)



        if (acc_val > best_acc_val):
            print("Best validation accuracy so far!")
            best_acc_val = acc_val
            iterations_since_best_acc = 0

            save_path = saver.save(sess,
"E:/Thesis_Simon_Serrarens/Python/tmp/testmodel_bench_simple.ckpt")

        else:
            iterations_since_best_acc += 1
            print("Epochs since last best validation accuracy:", iterations_since_best_acc)
            if iterations_since_best_acc == max_worse_iter:
                print("Early stopping")
                end = time.time()
                tot_time = end - start
                print("The training time for", epoch, " epochs is about:", tot_time)
                break


with tf.Session(config= session_conf) as sess:
    init.run()
    saver.restore(sess, "E:/Thesis_Simon_Serrarens/Python/tmp/testmodel_bench_simple.ckpt")
    acc_test = 0
    auc_test = 0
```

```
    n_batches = 0


    acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})

    print("The test accuracy of this model is ", acc_test)
```

Appendix I: The CNN model

Important: as in the MLP model, this script should read in the segments and labels first. This was cut out here to save space.

```
print("data fully prepared")

# In[ ]:

#Split again in train, test and validation sets

X_train = segments[0:1493180]
X_val = segments[1493180:1866475]
X_test = segments[1866475:]

y_train = labels[0:1493180]
y_val = labels[1493180:1866475]
y_test = labels[1866475:]


# In[ ]:

n_steps = 40
n_inputs = 36
channels = 1

conv1_fmaps = 32
```

```
conv1_ksize = 5
conv1_stride = 1
conv1_pad = "SAME"

pool1_fmaps = conv1_fmaps
pool1_stride = 2

conv2_fmaps = 64
conv2_ksize = 5
conv2_pad = "SAME"

pool2_fmaps = conv2_fmaps
pool2_stride = 2



drop_prob = tf.placeholder_with_default(0.0, shape=())

learning_rate = 0.00001
n_fc1 = 64
n_outputs = 2

with tf.name_scope("inputs"):
   X = tf.placeholder(tf.float32, shape=[None,n_steps, n_inputs], name="X")
   X_reshaped = tf.reshape(X, [-1, n_steps, n_inputs, channels])
   y = tf.placeholder(tf.int32, shape=[None], name="y")

conv1 = tf.layers.conv2d(inputs=X_reshaped, filters=32, kernel_size=[5,5],
                         padding=conv1_pad,
                         activation=tf.nn.relu, name="conv1")

pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2,2], strides=2)
```

```
conv2 = tf.layers.conv2d(pool1, filters=64, kernel_size=[5,5],
                         padding=conv2_pad,
                         activation=tf.nn.relu, name="conv2")

pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2,2], strides=2)

#Dense layers
pool2_flat = tf.reshape(pool2, [-1, 9*10*64])
dense = tf.layers.dense(inputs = pool2_flat, units = 10, activation = tf.nn.relu)
dropout = tf.layers.dropout(inputs = dense, rate = drop_prob)

dense2 = tf.layers.dense(inputs = dropout, units = 10, activation = tf.nn.relu)
dropout2 = tf.layers.dropout(inputs = dense2, rate = drop_prob)



#logits layer
logits = tf.layers.dense(inputs = dropout2, units = 2)

with tf.name_scope("train"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=y)
    loss = tf.reduce_mean(xentropy)
    optimizer = tf.train.AdamOptimizer(learning_rate = learning_rate)
    training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

with tf.name_scope("init_and_save"):
    init = tf.global_variables_initializer()
    saver = tf.train.Saver()
```

```
session_conf = tf.ConfigProto(
intra_op_parallelism_threads=12,
inter_op_parallelism_threads=12)


# In[ ]:


n_epochs = 100
batch_size = 20000
max_worse_iter = 3
best_acc_val = -9999
train_drop_prob = 0.75

start = time.time()
print("Training started")
with tf.Session(config = session_conf) as sess:
    init.run()
    for epoch in range(n_epochs):
        shuffled_idx = np.random.permutation(X_train.shape[0])
        n_batches = X_train.shape[0] // batch_size
        X_batches = np.array_split(X_train[shuffled_idx], n_batches)
        y_batches = np.array_split(y_train[shuffled_idx], n_batches)
        count = 0
        for X_batch, y_batch in zip(X_batches, y_batches):
            count += 1
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch, drop_prob: train_drop_prob})
            if (count % 10 == 0):
                acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
                print (count*batch_size, "Train accuracy = ", acc_train)
            if (count == 35):
                print("Midpoint reached, halfway epoch, calculating validation accuracy...")
```

```
            n_batches = 0
            acc_val = 0
            for c in range(X_val.shape[0] // batch_size):
                n_batches += 1
                offset = (c*batch_size)
                X_val_batch = X_val[offset:(offset + batch_size), :, :]
                y_val_batch  = y_val[offset:(offset + batch_size)]
                acc_val += accuracy.eval(feed_dict={X: X_val_batch, y: y_val_batch})



            acc_val = acc_val/n_batches
            print("Validation accuracy at half epoch:", acc_val)
            if (acc_val > best_acc_val):
                print("Best validation accuracy so far!")
                best_acc_val = acc_val
                iterations_since_best_acc = 0

                save_path = saver.save(sess,
"E:/Thesis_Simon_Serrarens/Python/tmp/testmodel_cnn_tf.ckpt")
            else:
                print("No improvement at this point, continue learning till full epoch.")


        n_batches = 0
        acc_val = 0
        print("End of epoch, calculating validation accuracy...")
        for c in range(X_val.shape[0] // batch_size): #X_val.shape[0] // batch_size)
            n_batches += 1
            offset = (c * batch_size)# % (X_val.shape[0] - batch_size)
            X_val_batch = X_val[offset:(offset + batch_size), : , :]
            y_val_batch = y_val[offset:(offset + batch_size)]
            acc_val += accuracy.eval(feed_dict={X: X_val_batch, y: y_val_batch})
        acc_val = acc_val/n_batches
```

```
        print("Epoch", epoch, "Train accuracy:", acc_train, "Validation accuracy", acc_val)


        if (acc_val > best_acc_val):
            print("Best validation accuracy so far!")
            best_acc_val = acc_val
            iterations_since_best_acc = 0

            save_path = saver.save(sess,
"E:/Thesis_Simon_Serrarens/Python/tmp/testmodel_cnn_tf.ckpt")

        else:
            iterations_since_best_acc += 1
            print("Epochs since last best validation accuracy:", iterations_since_best_acc)
            if iterations_since_best_acc == max_worse_iter:
                print("Early stopping")
                end = time.time()
                tot_time = end - start
                print("The training time for", epoch, " epochs is about:", tot_time)
                break


with tf.Session(config= session_conf) as sess:
    saver.restore(sess, "E:/Thesis_Simon_Serrarens/Python/tmp/testmodel_cnn_tf.ckpt")
    acc_test = 0
    n_batches = 0

    for c in range(X_test.shape[0] // batch_size): #X_val.shape[0] // batch_size)
        offset = (c * batch_size)# % (X_val.shape[0] - batch_size)
        X_test_batch = X_test[offset:(offset + batch_size), : , :]
        y_test_batch = y_test[offset:(offset + batch_size)]
```

```
        acc_test += accuracy.eval(feed_dict={X: X_test_batch, y: y_test_batch})
        n_batches += 1
    acc_test = acc_test/n_batches
    print("The test accuracy of this model is ", acc_test)
```

Appendix J: The CLSTM model

Important: as in the MLP model, this script should read in the segments and labels first. This was cut out here to save space.

```
print("data fully prepared")

# In[ ]:

#Split again in train, test and validation sets

X_train = segments[0:1493180]
X_val = segments[1493180:1866475]
X_test = segments[1866475:]

y_train = labels[0:1493180]
y_val = labels[1493180:1866475]
y_test = labels[1866475:]


# In[ ]:

n_steps = 40
n_inputs = 36
channels = 1
#LSTM
n_layers = 2
n_neurons = 10
```

```
conv1_fmaps = 32
conv1_ksize = 5
conv1_stride = 1
conv1_pad = "SAME"

pool1_fmaps = conv1_fmaps
pool1_stride = 2

conv2_fmaps = 64
conv2_ksize = 5
conv2_pad = "SAME"

pool2_fmaps = conv2_fmaps
pool2_stride = 2

drop = tf.placeholder_with_default(False, shape=())
learning_rate = 0.001

n_fc1 = 64
n_outputs = 2

keep_prob = tf.placeholder_with_default(1.0, shape=())



with tf.name_scope("inputs"):
   X = tf.placeholder(tf.float32, shape=[None,n_steps, n_inputs], name="X")
   X_reshaped = tf.reshape(X, [-1, n_steps, n_inputs, channels])
   y = tf.placeholder(tf.int32, shape=[None], name="y")

conv1 = tf.layers.conv2d(inputs=X_reshaped, filters=32, kernel_size=[5,5],
                         padding=conv1_pad,
```

```
                    activation=tf.nn.relu, name="conv1")

pool1 = tf.layers.max_pooling2d(inputs = conv1, pool_size=[2,2], strides = 2)

conv2 = tf.layers.conv2d(inputs=pool1, filters=64, kernel_size=[5,5],
                    padding=conv1_pad,
                    activation=tf.nn.relu, name="conv2")

pool2 = tf.layers.max_pooling2d(inputs = conv2, pool_size=[2,2], strides = 2)

#Dense layers
pool2_flat = tf.reshape(pool2, [-1, 64*10, 9])
lstm_cells = [tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons, activation = tf.nn.relu)
          for layer in range(n_layers)]
cells_drop = [tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob = keep_prob)
for cell in lstm_cells]
multi_cell = tf.contrib.rnn.MultiRNNCell(cells_drop)
outputs, states = tf.nn.dynamic_rnn(multi_cell, pool2_flat, dtype=tf.float32)

top_layer_h_state = states[-1][1]
logits = tf.layers.dense(top_layer_h_state, n_outputs, name="softmax")

with tf.name_scope("train"):
   xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=y)
   loss = tf.reduce_mean(xentropy)
   optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
   training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
   correct = tf.nn.in_top_k(logits, y, 1)
   accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

```
with tf.name_scope("init_and_save"):
    init = tf.global_variables_initializer()
    saver = tf.train.Saver()


session_conf = tf.ConfigProto(
intra_op_parallelism_threads=12,
inter_op_parallelism_threads=12)


# In[ ]:


n_epochs = 10
batch_size = 20000
max_worse_iter = 2
best_acc_val = -9999

start = time.time()
print("Training started")
with tf.Session(config = session_conf) as sess:
    init.run()
    for epoch in range(n_epochs):
        shuffled_idx = np.random.permutation(X_train.shape[0])
        n_batches = X_train.shape[0] // batch_size
        X_batches = np.array_split(X_train[shuffled_idx], n_batches)
        y_batches = np.array_split(y_train[shuffled_idx], n_batches)
        count = 0
        for X_batch, y_batch in zip(X_batches, y_batches):
            count += 1
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch, keep_prob: 0.5})
            if (count % 15 == 0):
                acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
```

```
            print (count*batch_size, "Train accuracy = ", acc_train)
        if (count == 35):
            print("Midpoint reached, halfway epoch, calculating validation accuracy...")
            n_batches = 0
            acc_val = 0
            for c in range(X_val.shape[0] // batch_size):
                n_batches += 1
                offset = (c*batch_size)
                X_val_batch = X_val[offset:(offset + batch_size), :, :]
                y_val_batch  = y_val[offset:(offset + batch_size)]
                acc_val += accuracy.eval(feed_dict={X: X_val_batch, y: y_val_batch})



            acc_val = acc_val/n_batches
            print("Validation accuracy at half epoch:", acc_val)
            if (acc_val > best_acc_val):
                print("Best validation accuracy so far!")
                best_acc_val = acc_val
                iterations_since_best_acc = 0

                save_path = saver.save(sess,
"E:/Thesis_Simon_Serrarens/Python/tmp/testmodel_clstm_eigen.ckpt")
            else:
                print("No improvement at this point, continue learning till full epoch.")

        n_batches = 0
        acc_val = 0
        print("End of epoch, calculating validation accuracy...")
        for c in range(X_val.shape[0] // batch_size): #X_val.shape[0] // batch_size)
            n_batches += 1
            offset = (c * batch_size)# % (X_val.shape[0] - batch_size)
            X_val_batch = X_val[offset:(offset + batch_size), : , :]
```

```
            y_val_batch = y_val[offset:(offset + batch_size)]
            acc_val += accuracy.eval(feed_dict={X: X_val_batch, y: y_val_batch})
        acc_val = acc_val/n_batches

        print("Epoch", epoch, "Train accuracy:", acc_train, "Validation accuracy", acc_val)



        if (acc_val > best_acc_val):
            print("Best validation accuracy so far!")
            best_acc_val = acc_val
            iterations_since_best_acc = 0

            save_path = saver.save(sess,
"E:/Thesis_Simon_Serrarens/Python/tmp/testmodel_clstm_eigen.ckpt")

        else:
            iterations_since_best_acc += 1
            print("Epochs since last best validation accuracy:", iterations_since_best_acc)
            if iterations_since_best_acc == max_worse_iter:
                print("Early stopping")
                end = time.time()
                tot_time = end - start
                print("The training time for", epoch, " epochs is about:", tot_time)
                break



with tf.Session(config= session_conf) as sess:
    saver.restore(sess, "E:/Thesis_Simon_Serrarens/Python/tmp/testmodel_clstm_eigen.ckpt")
    acc_test = 0
    n_batches = 0

    for c in range(X_test.shape[0] // batch_size): #X_val.shape[0] // batch_size)
```

```
        offset = (c * batch_size)# % (X_val.shape[0] - batch_size)
        X_test_batch = X_test[offset:(offset + batch_size), : , :]
        y_test_batch = y_test[offset:(offset + batch_size)]
        acc_test += accuracy.eval(feed_dict={X: X_test_batch, y: y_test_batch})
        n_batches += 1
    acc_test = acc_test/n_batches
    print("The test accuracy of this model is ", acc_test)
```

Appendix K: Code for the randomised grid search

Important: as in the MLP model, this script should read in the segments and labels first. This was cut out here to save space.

```
print("data fully prepared")

# In[ ]:

#Split again in train, test and validation sets

X_train = segments[0:1493180]
X_val = segments[1493180:1866475]
X_test = segments[1866475:]

y_train = labels[0:1493180]
y_val = labels[1493180:1866475]
y_test = labels[1866475:]



#Construct an LSTMclassifier that can be called with by the optimizer

session_conf = tf.ConfigProto(
intra_op_parallelism_threads=12,
inter_op_parallelism_threads=12)
```

```
class LSTMClassifier(BaseEstimator, ClassifierMixin):
    def __init__ (self, n_hidden_layers=2, n_neurons=100, optimizer_class = tf.train.AdamOptimizer,
                  learning_rate = 0.01, batch_size = 5000, activation=tf.nn.relu, keep_rate=0.5):
        self.n_hidden_layers = n_hidden_layers
        self.n_neurons = n_neurons
        self.optimizer_class = optimizer_class
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.activation = activation
        self.keep_rate = keep_rate
        self._session = None

    def _lstm(self, X):
        lstm_cells = [tf.contrib.rnn.BasicLSTMCell(num_units=self.n_neurons, activation = self.activation)
                      for layer in range(self.n_hidden_layers)]
        cells_drop = [tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob = self.keep_rate)
                      for cell in lstm_cells]
        multi_cell = tf.contrib.rnn.MultiRNNCell(cells_drop)
        outputs, states = tf.nn.dynamic_rnn(multi_cell, X, dtype=tf.float32)
        top_layer_h_state = states[-1][1]
        return top_layer_h_state

    def _build_graph(self, n_steps, n_inputs, n_outputs):
        #keep_prob = tf.placeholder(np.float32, shape=())
        X = tf.placeholder(tf.float32, [None, n_steps, n_inputs], name = "X")
        y = tf.placeholder(tf.int32, [None], name = "y")

        lstm_outputs = self._lstm(X)
```

```
    logits = tf.layers.dense(lstm_outputs, n_outputs, name="softmax")
    Y_proba = tf.nn.softmax(logits, name ="Y_proba")

    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")
    if self.optimizer_class == tf.train.MomentumOptimizer:
        optimizer = self.optimizer_class(learning_rate=self.learning_rate, momentum = 0.9)
    else: optimizer = self.optimizer_class(learning_rate=self.learning_rate)
    training_op = optimizer.minimize(loss)

    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

    init = tf.global_variables_initializer()
    saver = tf.train.Saver()
    self._training_op, self._accuracy = training_op, accuracy
    self._init, self._saver = init, saver
    self._X, self._y = X, y
    self._Y_proba = Y_proba
    self._loss = loss


def close_session(self):
    if self._session:
            self._session.close()

def _get_model_params(self):
    with self._graph.as_default():
        gvars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES)
    return {gvar.op.name: value for gvar, value in zip(gvars, self._session.run(gvars))}

def _restore_model_params(self, model_params):
```

```
        gvar_names = list(model_params.keys())
        assign_ops = {gvar_name: self._graph.get_operation_by_name(gvar_name + "/Assign")
                      for gvar_name in gvar_names}
        init_values = {gvar_name: assign_op.inputs[1] for gvar_name, assign_op in
assign_ops.items()}
        feed_dict = {init_values[gvar_name]: model_params[gvar_name] for gvar_name in
gvar_names}
        self._session.run(assign_ops, feed_dict=feed_dict)

    def fit(self, X, y, n_epochs = 10, X_valid=None, y_valid=None):
        self.close_session()

        self.classes_ = np.unique(y)
        n_inputs = 36
        n_steps = 40
        n_outputs = 2
        best_loss = np.infty
        best_params = None

        self._graph = tf.Graph()
        with self._graph.as_default():
            self._build_graph(n_steps, n_inputs, n_outputs)

        max_worse_iter = 2
        iterations_since_best_acc = 0



        self._session = tf.Session(graph=self._graph, config = session_conf)
        with self._session.as_default() as sess:

            self._init.run()
```

```
for epoch in range(n_epochs):
    shuffled_idx = np.random.permutation(X_train.shape[0])
    n_batches = X_train.shape[0] // self.batch_size
    X_batches = np.array_split(X_train[shuffled_idx], n_batches)
    y_batches = np.array_split(y_train[shuffled_idx], n_batches)
    count = 0
    for X_batch, y_batch in zip(X_batches, y_batches):
        count += 1
        sess.run(self._training_op, feed_dict={self._X: X_batch, self._y: y_batch})

    #Calculate validation accuracy
    if X_valid is not None and y_valid is not None:
        loss_val, acc_val = sess.run([self._loss, self._accuracy],
                                     feed_dict={self._X: X_valid, self._y: y_valid})

        print("Epoch", epoch,"Validation accuracy", acc_val)



        if (loss_val < best_loss):
        #print("Best validation accuracy so far!")
            best_loss = loss_val
            iterations_since_best_acc = 0
            best_params = self._get_model_params()
        else:
            iterations_since_best_acc += 1
        #print("Epochs since last best validation accuracy:", iterations_since_best_acc)
            if iterations_since_best_acc == max_worse_iter:
                print("Early stopping")

            #print("The training time for", epoch, " epochs is about:", tot_time)
                break
```

```
            if best_params:
                self._restore_model_params(best_params)
            return self

    def predict(self, X):
        class_indices = np.argmax(self.predict_proba(X), axis=1)
        return np.array([[self.classes_[class_index]]
                         for class_index in class_indices], np.int32)

    def predict_proba(self, X):
        if not self._session:
            raise NotFittedError("This %s instance is not fitted yet" % self.__class__.__name__)
        with self._session.as_default() as sess:
            return self._Y_proba.eval(feed_dict={self._X: X})

print("all functions defined")
print("testing a mock model with 10 epoch ,testing only 1 model, based on what I tested already
before")

param_distribs = {
"n_neurons": [36, 72, 108],
"batch_size": [5000, 10000, 20000],
"learning_rate": [0.01, 0.001, 0.0001],
"n_hidden_layers": [2,3,4],
"optimizer_class": [tf.train.AdamOptimizer, tf.train.MomentumOptimizer,
tf.train.RMSPropOptimizer],
"keep_rate": [0.3, 0.5, 0.7]
}

rnd_search = RandomizedSearchCV(LSTMClassifier(), param_distribs, n_iter = 40,
random_state = 42, verbose = 2)
fit_params = {"X_valid": X_val, "y_valid": y_val}
```

```
print("fitting started")
rnd_search.fit(X = X_train, y = y_train, X_valid = X_val, y_valid = y_val)
print("Completed fitting. Best parameters are:")
print(rnd_search.best_params_)

print("using this model to predict accuracy:")
y_pred = rnd_search.predict(X_test)
acc = accuracy_score(y_test, y_pred)

print("Test accuracy for this model is:" ,acc)
print("Try to Determine F1 score of the model")
try:
   f1 = f1_score(y_test, y_pred)
   print("the F1 of this model is", f1)
except: print("didn't work")
print("model terminated normally")
```