UNIVERSITY OF TARTU
Institute of Computer Science

Vahur Madisson

Forecasting and Trading Financial Time Series with LSTM Neural Network

Conversion Master in IT

Master Thesis (15 ECTS)

Supervisors: Toomas Raus, Ph.D.,
Meelis Kull, Ph.D.

Tartu 2021

Forecasting and Trading Financial Time Series with LSTM Neural Network

Abstract: The growing importance of data science and the development of machine learning allows to implementation of the algorithms created in recent decades with new capable technologies. Machine learning methods can challenge statistical methods of forecasting when applied in financial time series, as such data may exhibit non-linear characteristics. The objective of the thesis is to present a theoretical introduction and practical steps to construct, test, and implement forecasting methods on the stock market index, using artificial intelligence algorithm called long short-term memory (LSTM) neural network. The relevant trading strategy is developed to implement the model predictions. The empirical study focuses on finding the best configuration of the LSTM model to enhance the forecasting ability, using Keras library in Python programming language. The results are assessed in terms of forecast accuracy measures and profitability when applying relevant trading strategy and compared against selected benchmark methods. Results demonstrate that LSTM forecast accuracy is competitive and trading results outperform compared to selected benchmarks methods.

Keywords: Artificial neural networks, time series analysis, forecasting, trading strategy
CERCS research specialization: Artificial Intelligence (P176)

Finantsaegridade prognoosimine ja kauplemine LSTM tehisnärvivõrguga

Lühikokkuvõte: Andmeteaduse kasv võimaldab masinõppe jõudsat arengut ning uute tehnoloogiatega saab tõhusamalt rakendada viimastel aastakümnetel välja töötatud algoritme. Masinõppe meetodid võivad edestada prognoosimise statistilisi meetodeid, eriti kui andmetel nagu näiteks finantsaegridadel on mittelineaarseid omadusi. Lõputöö eesmärk on esitada teoreetiline sissejuhatus ja praktilised sammud prognoosimeetodite juurutamiseks finantsaegridadele aktsiaturu indeksi näitel, leides võimalikult tulemuslikku seadet tehisintellekti algoritmimeetodi - pikaajalise lühimälu (LSTM) närvivõrgule. Prognooside põhjal kauplemiseks on koostatud asjakohane kauplemisstrateegia. Empiiriline uuring keskendub LSTM mudeli prognoosimisvõime parimale rakendamisele ja tulemustele võrreldes baasmeetoditega kasutades Pythoni programmeerimiskeele Kerase paketti. Uuringu tulemused aitavad hinnata praktilist kasu LSTM kaudu finantsaegridade prognoosimisel ja nende põhjal kauplemisel. Tulemusi hinnatakse prognooside täpsuse ja kauplemisreeglitega saavutatud kasumlikkuse osas. Tulemuste põhjal võib väita, et LSTM mudeli prognositulemused on konkurentsivõimelised ja kasumlikud ning edestavad baasmeetodeid.

Võtmesõnad: tehisnärvivõrgud, aegridade analüüs, prognostika, kauplemisstrateegia
CERCS teaduseriala: Tehisintellekt (P176)

Contents

Introduction

The algorithms to forecast financial time series have been around for several decades. Until recently, when the computational resources and programming tools have become more available and the machine learning in image recognition and natural language processing has done breakthroughs, it has facilitated the research of more complex algorithms in financial time-series.

The forecasting of financial time series with machine learning is of interest to several fields of study. Academics of finance and statistics is mainly concerned with observing the efficient market hypothesis, as machine learning allows to discover inefficiencies in the financial markets. Especially, as machine learning methods allow non-linear characteristics without strict postulations about data, it may offer improvement to statistical methods that require such assumptions. It is important for studying relationships and forecasts of financial asset prices, which are primarily non-linear and non-stationary. The research can also be used in the investing profession to improve the investment results for example in investment funds. The computer science field provides the universal infrastructure for time-series-related data science and facilitates the research objectives.

The thesis focuses on the particular subfield of machine learning models called artificial neural networks, which have achieved very good results in solving problems in various fields, such as image recognition and natural language processing. Neural networks consist of computational units called neurons, which are connected as a network. They process the information received from input features in the form of the financial time series, and the network is tasked to output the target or the time-series prediction in this case. During the training, neurons get the task to modify their connected weights to produce the target prediction with minimal error to true target. The process of learning is run multiple times to minimize the error. The final state of the network model is stored and executed to output the target predictions in out-of-sample tests.

The first objective of the thesis is to present the theoretical foundations and practical steps to construct, test, and implement neural network forecasting methods on a stock market index. The chosen artificial neural network algorithm is called long short-term memory (LSTM), which has shown promising results in various research studies.

The second objective is to assess how the LSTM model performs compared to other methods used in quantitative finance and machine learning. Selected benchmark models are explained, followed by measuring and presenting the results in empirical study and discussing the comparisons.

The third objective is to explore and test LSTM model settings to improve the results of the performance metrics chosen for empirical study. The number of hyperparameter configurations is tested, both setting them manually and automatically. Various research has been published on financial time-series forecasting with artifical neural networks, being

primarily focused on predictions using the asset's historical prices or the transformations of these prices. The thesis explores whether introducing related economical, financial, and technical analysis data can improve the performance results. The neural network results are compared against set of benchmark models to assess the attractiveness of the LSTM models.

The fourth objective of the thesis is to construct a relevant trading strategy for implementing the neural network forecasts. The trading strategy aims to execute model predictions efficiently and with amount of capital invested comparable to buy and hold investment strategy.

The thesis is structured so that Section 1 provides fundamental concepts and explain artificial neural network architecture and LSTM model. Section 2 prepares and details requirements and settings for empirical study. It includes the time series forecasting methodology, selecting and explaining the benchmark methods, describing the data preprocessing and data transformation for the neural network modeling, and introducing methods to improve neural network performance. Section 2 also defines the performance evaluation framework for assessing the neural network forecasting ability. The trading strategy performance metrics and setup of trading strategy are explained. Section 3 describes the empirical study carried out to achieve the research objectives. In this section, the dataset and technology is detailed, the experiments are explained, and the findings are presented, concluding with the discussion of results.

1 Artificial Neural Networks

1.1 Fundamental Concepts

In this section book by Haykin (2009, p.1-15) is used if not referred otherwise. Artificial Neural Network (ANN), commonly known as neural network (NN) is a machine designed to model the human brain in performing a particular task through learning. It is a simplification of biological nerve cell which, as illustrated in Figure 1 receives the input in the form of signals in the dendrites of a neuron and transforms the signal to outputs at axon terminals.

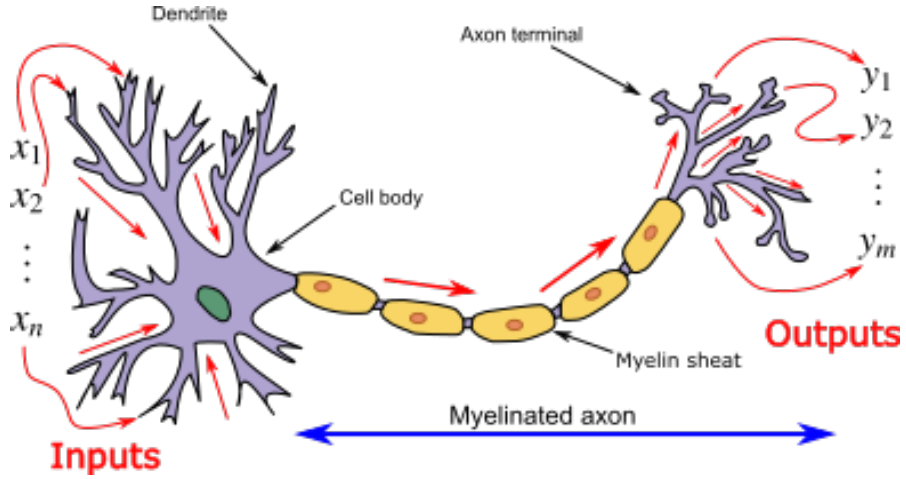


Figure 1. „Neuron and myelinated axon, with signal flow from inputs at dendrites to outputs at axon terminals“ by prof. Loc Vu-Quoc, licenced under CC BY- SA 4.0.

As a model of the biological nerve cell, neural network is designed as a parallel distributed processor made of simple processing units, which consist of 1) interconnections of simple computational cells referred to as neurons or as nodes that carry the network signals, and 2) of the weights attached to the neurons, that contribute as the neuron's connection strength and act as storage of experiential knowledge through connections.

One of the first ANN models was proposed by (Rosenblat, 1958) and is called perceptron, which is often used as a starting point to explain modern ANN principles. A perceptron example in Figure 2 takes three binary inputs and produces a single binary output.

Rosenblat introduced weights as real numbers to express the importance of the respective inputs. The neuron output, 0 or 1, that biologically corresponds to states of neuron not firing or firing is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than a threshold value. The perceptron learns the classification task by changing the weights so that the error between the target and the output is minimized. After moving

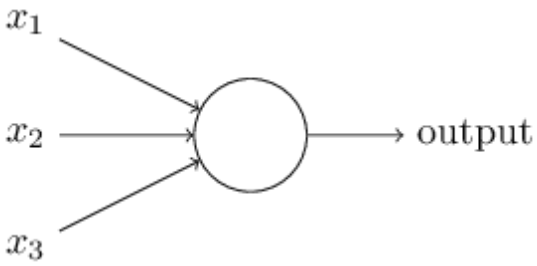


Figure 2. Perceptron

the threshold to the other side of the inequality and replacing it by the bias, the perceptron rule can be written:

$$\text{Output } y = \begin{cases} 0 & \text{if } \sum_j w_j x_j + b \leq 0 \\ 1 & \text{if } \sum_j w_j x_j + b \geq 0. \end{cases}$$

The bias in biological terms represents how easy it is to get the perceptron to fire, so that larger bias value makes easier to output 1. Moving further from perceptron to single-layer neural network, the additional model feature called activation function is added to the neuron model. Such a non-linear model of a neuron labeled k, which is an oversimplified model of biological neuron in Figure 1, has inputs, weights, a bias, a summation and an activation function (Figure 3). The bias has been decomposed into external fixed input $x_0 = +1$ with weight w_{k0} so that $b_k = w_{k0}x_0 = w_{k0}$.

The induced local field of a neuron is expressed as:

$$v_k = \sum_{j=1}^{m} w_{kj} x_j + b_k.$$

The activation function's purpose is to stabilize the output value for the changes in input or weight values and facilitate the model learning process. Compared to perceptron, which can have output 0 or 1, the inclusion of activation function adds the benefit of smoothing out the neuron output to have any real number in interval defined by activation function. The smoothness allows small changes Δw_j in the weights and Δb in the bias to produce a small change $\Delta\, y$, so that $\Delta\, y$ can be approximated to be linear function of $\Delta\, w_j$ and $\Delta\, b$. This linearity makes it easy to choose small $\Delta\, w_j$ and $\Delta\, b$ to achieve any desired $\Delta\, y$ (Nielsen, 2015), which is beneficial property during ANN learning process. Depending on the problem to be solved with ANN, the activation function $\varphi(v)$ is often preferred to transform the neuron output to be any real number in interval $(0, 1)$, which

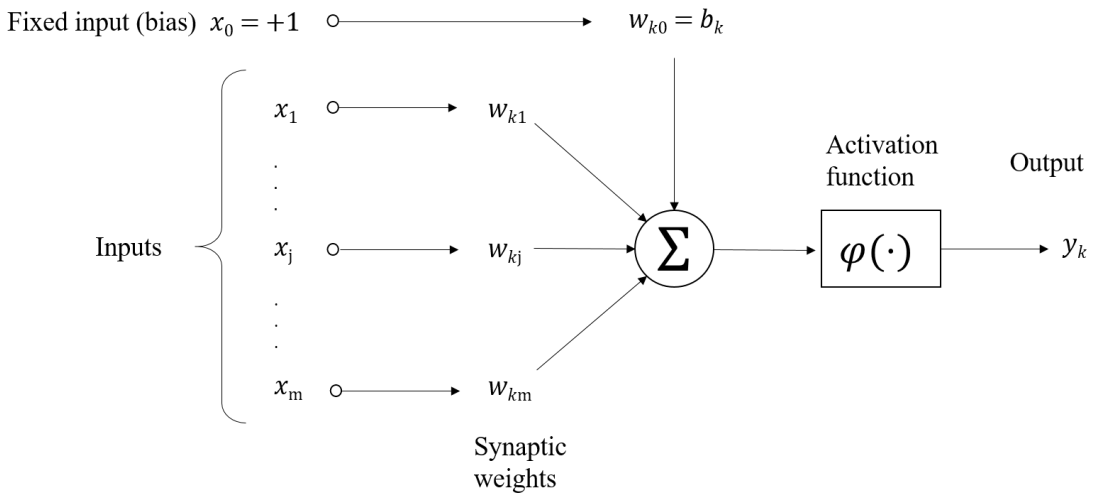


Figure 3. A non-linear model of a neuron

is then sigmoid (sometimes called logistic) function as follows:

$$\varphi(v_k) = \frac{1}{1 + exp(-v_k)}.$$

Alternatively, if preferred to any real number in the interval $(-1, 1)$, which sometimes may yield computational benefits over the sigmoid function, then the tangent function is used:

$$\varphi(v_k) = tanh(v_k).$$

Finally, the output of a neuron is expressed as follows:

$$y_k = \varphi(v_k) = \varphi\left(\sum_{j=0}^{m} w_{kj} x_j\right).$$

1.2 Neural Network Architectures

This section is based on (Goodfellow et al., 2016).

1.2.1 Feedforward Networks

The extension of the perceptron is the multilayer perceptron (MLP), which is designed to overcome the single-layer perceptron limitation to classify only linearly separable patterns. It is also called a feedforward neural network because information flows from x through the intermediate computation used to define f and finally to the output y. The goal of MLP is to approximate some function f^* to map input x to a category y, so that $y = f^*(x)$. During neural network training, the $f(x)$ is driven to match $f^*(x)$. There are no feedback connections from the output that is fed back into itself.

Feedforward architecture (Figure 4) has at least one hidden layer of neurons. The overall length of the chained layers represents the depth of the network model, which also has given name to deep learning.

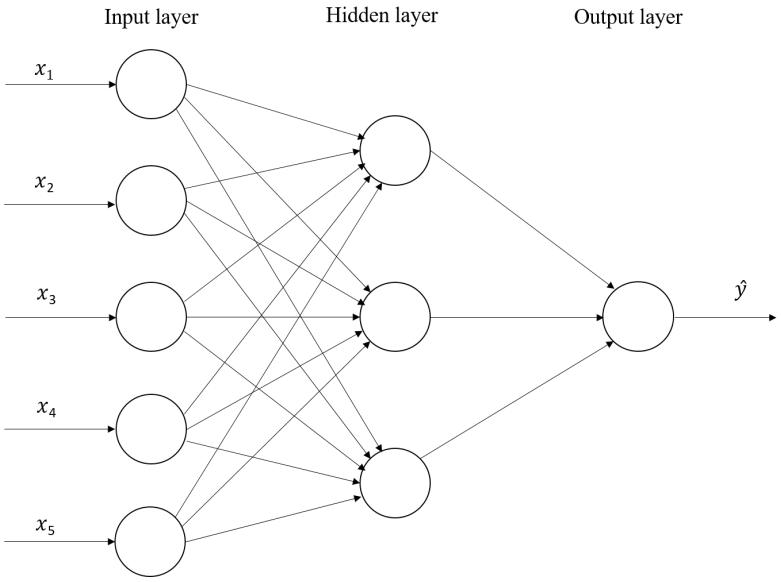


Figure 4. Example of a feedforward network with one hidden layer with 3 neurons

The learning algorithm uses the layers to approximate $f(x)$ towards $f^*(x)$. The intermediate layers are called hidden layers, because the training data does not show output each layer's output. Each unit in the layer resembles a neuron. It receives input from other subsequent units and computes its own activation value, which then serves as a new feature in the network going forward. The multilayer concept enables more flexible

separation of the values and therefore better discovery of patterns than single-layer networks, (see Goodfellow et al., 2016, Ch. 6).

1.2.2 Recurrent Neural Networks

After adding feedback connections to the feedforward network, it is called recurrent neural network or RNN (see Goodfellow et al., 2016, Ch.10), as illustrated in the computational graph in Figure 5.

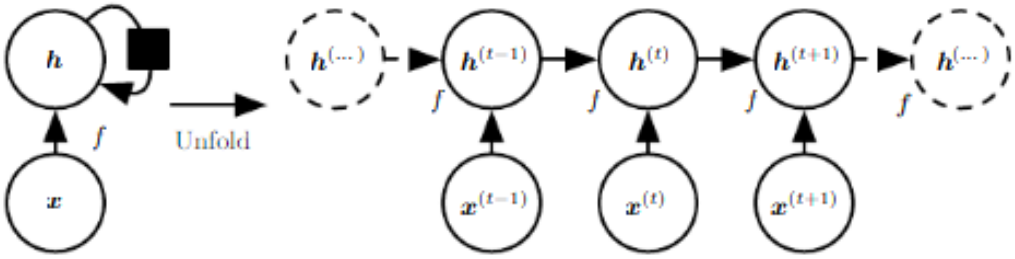


Figure 5. Recurrent neural network with no outputs that processes information from x and incorporate it into state h that is passed forward through time t. Retrieved from (Vicente, 2019).

RNN specializes in processing sequences with fixed and, in some cases of variable length, which are often practicable in domains of machine learning such as text, speech, stock markets, DNA sequence, etc. The specific idea of RNN is to share parameters across the network that allows generalization of data sequences beyond model training data. This design aspect is practicable for time series that provide sequence of data. Also, for financial time series where the current value of a variable can influence the variable value in a future time step, the RNN allows approximation of such influences across sequences by sharing the parameters. Traditional feedforward network has separate parameters θ for each input feature, while recurrent neural network shares the same parameters across several time steps. The unfolded recurrence after t steps can be represented with a function $g^{(t)}$:

$$h^{(t)} = g^{(t)}(x^{(t)}, x^{(t-1)}, x^{(t-2)}, \ldots, x^{(2)}, x^{(1)}) = f(h^{(t-1)}, x^{(t)}; \theta).$$

1.2.3 Back-propagation

Feedforward network takes input x, which propagates up to hidden units in each layer, so that information flows forward through the network, then produces output $\hat{y}$ and the loss $L(\hat{y}, y)$ associated with (x, y), as in (Goodfellow et al., 2016, Ch. 6). This process

is called forward propagation. The feedforward network model is trained with a learning algorithm using forward propagation to approximate $f(x)$ towards $f^*(x)$ to minimize $L(\hat{y}, y)$. The learning process can be improved by using the back-propagation algorithm, which allows the information to flow backward through the network by computing the gradients of parameters - weights and biases, in each step. While back-propagation is merely a computation of gradients, the learning algorithm called gradient descent, introduced in Section 1.2.4, is performing the learning process during training using the back-propagation method. The back-propagation is recursively applying the chain rule of calculus to compute the derivatives of functions in each step of the neural network chain. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that the derivative of z with respect to x can be expressed as

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}.$$

1.2.4 Stochastic Gradient Descent

The learning algorithms involve optimization to minimize or maximize some function $y = f(x)$ by altering the x. In terms of neural network, we minimize the objective function also called cost function or loss function. The derivative $f'(x)$ gives the slope of $f(x)$ at the point x. It also helps to scale the size of the step ϵ also called learning rate, to obtain the corresponding change in the output $\hat{y}$ so that $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$. The derivative implies how to change x to minimize $L(\hat{y}, y)$. As $f(x) - \epsilon sign(f'(x))$ is less than $f(x)$ for a small ϵ, then $f(x)$ can be reduced by moving x in small steps with the opposite sign of the derivative. This technique is called gradient descent. When changing the weights in the direction of negative gradient, the cost is being reduced by gradient descent, as illustrated in Figure 6. The back-propagation algorithm results a gradient matrix of all weights in neural network for the cost function.

Gradient descent learning algorithm is used not only with neural networks but also in many machine learning models. While the training data set becomes large, it also becomes computationally expensive to use gradient descent. The stochastic gradient method obtains the unbiased estimates of the gradients based on a small set of uniformly drawn samples from training data called minibatches. The estimate of gradient $g = f'(x)$ follows the estimated gradient downhill

$$\theta \leftarrow \theta - \epsilon g.$$

The learning rate is a crucial parameter that defines the speed of the model convergence to minimum cost, when too small value of ϵ causes slow convergence while too large value of ϵ can make learning unstable. The convergence is reached when the gradient becomes a small value close to zero. The stochastic gradient descent algorithm as an optimization technique can be improved further with the addition of the momentum of

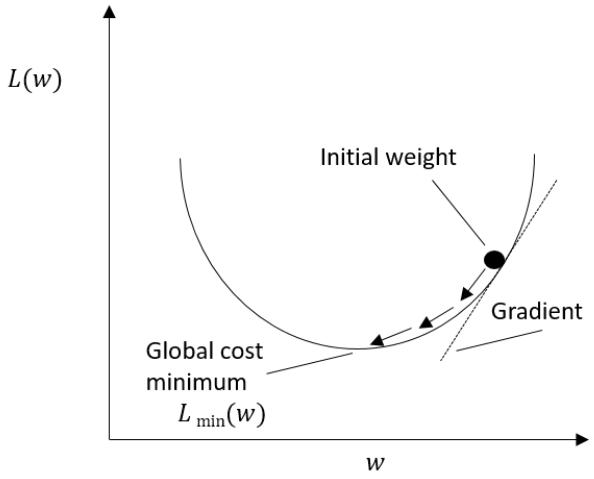


Figure 6. Gradient descent. An illustration of how the gradient descent algorithm uses the derivative to follow the function downhill to a minimum. (Lecture 5, Vicente, 2019)

the successive gradients. The size of the step previously determined by learning rate is adjusted by how large and how aligned the sequence of gradients is. The step size becomes larger when successive gradients point more towards the same direction, thereby speeding up the convergence.

1.2.5 Objective Functions

The algorithms in machine learning try to solve the task by minimizing or maximizing a given function. If the minimizing function is used, then it is also called the loss function. The choice of objective function in machine learning depends on the type of problem, typically classification or regression. The thesis focuses on prediction as a regression problem. This section presents only most common loss functions, as explained by (Sam Lau, 2019, Ch. 10) and (Trevor Hastie, 2017, p. 219-223) and the variants with attractive propositions to our task, all presented and tested in the empirical section of the thesis. Loss function is defined to measure which parameter setting of the model and its output $\hat{y}$ is best to minimize the loss L as $L_{min}(\hat{y}, y)$. The loss function is what neural network model is learning to minimize during its updates and thereby achieve its target within the task of its defined problem. The first loss function is mean squared error (MSE) and is calculated as

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

where y_i is true value and $\hat{y}_i$ is output value.
The second cost function is mean absolute error (MAE) and is calculated as

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

The comparison of MSE and MAE formulas reveals that since MSE has a quadratic error term it penalizes the loss more than MAE. Therefore MSE is preferred loss function when we consider outliers being systematically present in the data, as can be the case in financial markets. As in the (Sam Lau, 2019), MSE is also easier to differentiate, it always has a derivative, which is important to enable learning process in the gradient descent.

The Huber loss is the third function, which combines both MSE and MAE to be both differentiable and robust to outliers. The Huber loss achieves this by performing as MSE for errors close to minimum loss and switching to the absolute loss for errors far from a minimum loss. The additional parameter α sets the point where the Huber loss transitions from the MSE to the MAE. The Huber loss function is defined as follows:

$$L_\alpha(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n}\begin{cases}\frac{1}{2}(y_i - \hat{y}_i)^2 & \text{if } |y_i - \hat{y}_i| \leq \alpha \\ \alpha(|y_i - \hat{y}_i| - \frac{1}{2}\alpha) & \text{if otherwise.}\end{cases}$$

The loss function variant that might be considered for the empirical study of this thesis is called the stock loss function, described in (Davidson-Pilon, 2016). It is a regression function, that adds penalizing term to the loss if the prediction output predicts the wrong direction for the output. In our problem setting it might be important, as when the model predicts positive (negative) return for the financial instrument, when it is actually negative (positive) return, such loss has a higher impact on the trading performance if the predicted output is accurate direction. The stock loss function is defined as follows, where α controls the penalty amount,

$$L_\alpha(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n}\begin{cases}(y_i - \hat{y}_i)^2 & \text{if } (y_i \cdot \hat{y}_i) > 0 \\ (y_i - \hat{y}_i)^2 + \alpha \cdot \hat{y}_i^2 & \text{if } (y_i \cdot \hat{y}_i) < 0.\end{cases}$$

1.3 Long Short-Term Memory Model

This section is mostly based on (Goodfellow et al., 2016, p.397-400), (Hochreiter and Schmidhuber, 1997) and (Olah, 2015). As introduced in Section 1.2.2 , the RNN-s are sharing the parameters so that each output is a function of a previous output with the same update rule. Sharing the parameters and updates is essential in order to generalize to sequences not seen during the training. While plain RNN produces an output at each

time step, the version of RNN used in time series training and in the empirical study in the thesis, provide single output $o^{(t)}$ for a sequence of data, as in Figure 7. The gradient of $o^{(t)}$ is back-propagated through time from downstream modules to minimize the $L^{(t)}$.

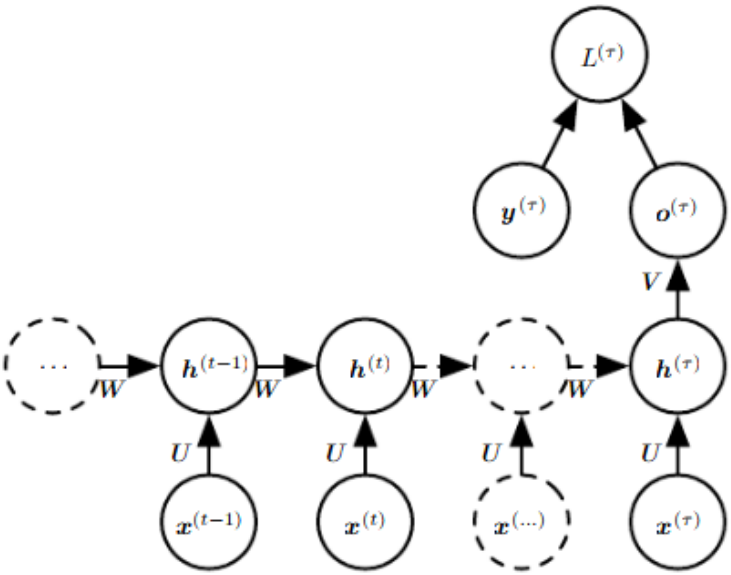


Figure 7. Time-unfolded recurrent neural network with a single output $o^{(t)}$ at the end of the sequence.

The problem with such RNN is that activations and gradients propagated over many stages tend to vanish often or rarely explode due to sharing parameters across the stages. Also, even if the parameters lead to stable RNN, the long-term dependencies get exponentially smaller weights than short-term. This means network cannot store such long-term dependencies in memory and has difficulties learning dependencies. The (Hochreiter and Schmidhuber, 1997) article proposed the RNN with an appropriate gradient-based learning algorithm called Long-Short-Term Memory (LSTM) as a solution.

In simple RNNs, within the chain of repeating cell modules, the new cell state h_t is produced by $tanh$ applied on the previous cell state h_{t-1} and new input x_t as in diagram in Figure 8.

LSTM module as in Figure 9 has additional gate layers to modify both cell state C_t and output h_t, to pass through only most relevant information. Three gates controls the information: forget gate, input gate and output gate.

The forget gate determines which part of the previous cell state C_{t-1} is relevant to store, and the rest is forgotten. It accomplishes this by applying a sigmoid function to the input

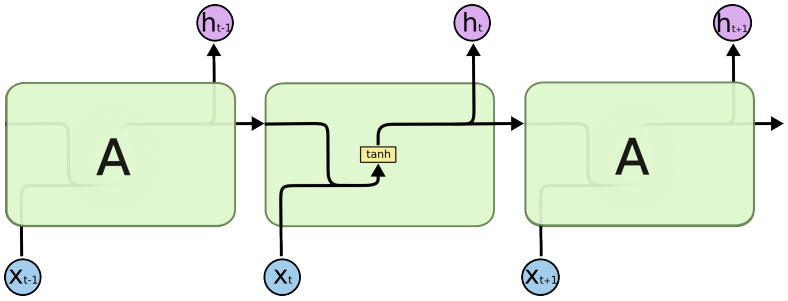


Figure 8. The repeating module in simple RNN. Retrieved from (Olah, 2015).

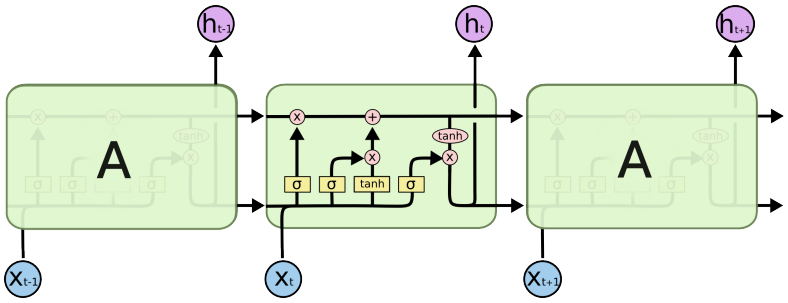


Figure 9. LSTM module. The top horizontal line is the cell state flow, and the bottom lines are hidden layers regulating the information update in the cell state. Retrieved from (Olah, 2015)

value h_{t-1} and x_t, and outputs a value between 0 and 1 for each value in C_{t-1}. The forget gate value is calculated as follows:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f),$$

where W_f is weight matrix, h_{t-1} is hidden layer vector containing outputs of all LSTM cells, b_f is bias vector, x_t is the input at time t and $\sigma(\cdot)$ is a sigmoid function. In the next step, input gate decides what the new information being updated to cell state is. It does it by 1) deciding which values are updated, and is calculated as follows:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i),$$

and 2) decides which are the new candidate values to cell state, calculated as follows:

$$\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C).$$

Then, the input gate combines the above two to provide new information update $i_t\tilde{C}$.

Then new information update is combined with remained information from past cell state to result in the new cell state, calculated as follows:

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t.$$

Finally, the output gate decides what the output is and does it in two steps. At first, the sigmoid function is filtering the input values similar to f_t and i_t in order to decide which cell state values are going to be in output, as follows:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o),$$

and then calculates the final output values by multiplying o_t by tanh squashed values of cell state C_t,

$$h_t = o_t tanh(C_t).$$

The prediction value h_t and cell state C_t are forwarded to next LSTM cell at time $t + 1$.

2 Time Series Forecast Modeling Methodology

2.1 Benchmark Methods

In the thesis, we are predicting the stock index return for the next day, where the index one-period return is y_{t+1} for which the prediction is $\hat{y}_{t+1}$. We define a regression problem to forecast $\hat{y}_{t+1}$ by minimizing training data error to y_{t+1}. After the prediction is made, the prediction is classified to positive or negative predicted return, and the stock market index is traded based on predicted class.

Let the index closing price s at time t be s_t, then continuously compounded daily return for the index is

$$y_t = ln(s_t/s_{t-1}).$$

The continuous compounded return, in other words logarithmic return $ln(s_t/s_{t-1})$ is used instead of arithmetic return $(s_t/s_{t-1} - 1)$, following a general practice in quantitative finance. The logarithmic returns are symmetric, so that positive and negative return of equal size cancels out. Logarithmic and arithmetic returns are equal only when the return is zero and the difference is smaller when the return is smaller, so that $ln(s_t/s_{t-1}) \approx s_t/s_{t-1} - 1$ if $s_t/s_{t-1} \approx 1$.

2.1.1 Naïve Method

For setting the first baseline forecasting method, we have chosen the naïve method that uses change over time as forecast, as explained by (Hyndman and Athanasopoulos, 2018). As we use y_t as the return over day t, then return forecast for $t + 1$ is given by

$$\hat{y}_{t+1|t} = ln(s_t/s_{t-1}).$$

As a result, the method provides a new forecast for each day. Naïve methods are generally considered optimal for random walk forecasts. Assuming that equity markets generally follow random walk, naïve method is considered an appropriate baseline model in the thesis.

2.1.2 Empirical Benchmark Method

A metric that is considered for comparison of trading systems is the accuracy of the forecast. As proposed in (Krause and Fairbank, 2020), the empirical model suggests forming a simple benchmark that either forecasts always positive returns, or always forecasts negative returns. The choice between using the positive or negative return forecast is motivated by the assumption that the overall trend of the time series is known to the forecaster, as might be the case with broad equity markets over the long-term

horizon. Also, if by pure chance the forecasting system happens to have a forecasting bias consistent with the actual market trend, then the forecasting system would look effective, but might not be if the market trend is taken into account. For that reason, baseline forecasted return $\bar{y}$ is set to be the geometric mean of the daily returns during the training period as

$$\bar{y} = (\prod_{i=1}^{n}(1 + y_i))^{1/n} - 1,$$

where n is the number of daily returns in the training period.

We have used the framework proposed in (Krause and Fairbank, 2020), but adjusted it with the assumption, that simple benchmark forecasts the return based on the mean of the training period returns instead of using the test period or combined training and test period mean. Thus the baseline data is without forward-looking bias. Also, as further demonstrated in the thesis, the testing consists of multiple test periods. Therefore the likelihood of the neural network accidentally approximating the stock index trend for the full test period is reduced. During the test period the benchmark method assumes forecasted return for each day is the geometric mean of the training period daily returns, and then test period performance metrics are calculated.

2.1.3 ARIMA Model

ARIMA is the Auto-Regressive Integrated Moving Average model, applied to time-series data to understand or predict the data. ARIMA can be viewed as a filter to separate the signal from the noise and use that signal to forecast future data points. ARIMA(p, d, q) model approximates time-series by autoregressive process of order p and moving average process of order q. In case of non-stationary series it is differenced by d times to make it stationary. The time series is stationary when mean, variance and covariance are constant over time.

Autoregressive process of order p forecasts the series by a linear combination of series previous p values, as

$$y_t = \mu + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + \varepsilon_t,$$

where μ is constant, $\phi_1, ..., \phi_p$ are parameters and $\varepsilon_t, t = 1, 2, ...,$ are are uncorrelated, normally distributed errors with a mean of 0 and variance σ^2.

Moving average process of order q forecasts the series by linear combination of previous q forecast errors. As in (Hyndman and Athanasopoulos, 2018), y_t can be thought of as a weighted moving average of the past forecast errors, expressed as

$$y_t = c + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \cdots + \theta_p \varepsilon_{t-q},$$

where c is constant, $\theta_1, ..., \theta_p$ are parameters.

When the initial time series is not stationary, then it is made stationary by differencing with order d, and the differenced series y'_t can be modeled with the ARIMA model

$$y'_t = c + \phi_1 {y'}_{t-1} + \cdots + \phi_p {y'}_{t-p} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \cdots + \theta_p \varepsilon_{t-q}.$$

2.1.4 Gradient Boosting Decision Trees

This section is mostly based on (Flach, 2017) and introduces the baseline method used in the thesis. We use the decision tree method, specifically its enhanced method called gradient boosting, in the empirical study. It is a machine learning model that requires almost no tuning and often delivers good results.

The decision tree is using the tree-like model to predict class labels. Decision tree presents the distribution of labels across all features. It splits the features by labels conjuctively and results in the formation of nodes that connect features according to label. For each feature's label the tree is constructed so that one feature is put on the root node and is split into its possible classes in case of classification problem. For these possible classes successive feature with its labels are assigned. The assignment of labels follows the mapping of feature's labels to the target. The splitting is repeated until all features are represented under the given root feature. The splitting follows specific rules that first divide data into subsets, build a tree for each subset and then combine subtrees into a single tree. The split is implemented recursively until the decision tree is found with the best purity. Feature trees are constructed to represent the conjunctive concepts in the hypothesis space. The learning problem is to decide which of the concepts is best to solve the given task, as expressed by the final decision tree.

Random forest is an ensemble decision tree method that constructs multiple decision trees. It builds each tree from a different random sample of features. It applies bootstrapping aggregation for each tree, which is building multiple trees again from repeatedly resampled training data, so that encourages diversity in the model.

Boosting is ensemble decision tree learning method to create diverse training sets from data, that uses optimization to minimize the objective function. Ensemble methods sum the prediction of different decision trees. Boosting algorithm creates ensemble by adding trees that give larger weight to previously misclassified examples, that improves the model by reducing the bias when minimizing the loss. Gradient boosting uses gradient descent algorithm to minimize loss when new tree models are added, while it supports both classification and regression predictive modeling. Finally, the learning process arrives to model setting that is best fitted to the training data and that is used to predict the target. As the main focus of the thesis is on Neural Networks using gradient descent, the gradient boosting is considered an appropriate baseline model in the Empirical study.

2.2 Neural Network Methods

2.2.1 Data arrangement for neural network in Empirical study

First, we split the values of available features, also called input variables, and target variables into sequences of study periods. Then each study period is separated to training data, validation data, and testing data. Training data is used to train the neural network model to fit the features to target. Validation data is used to predict based on trained model and on new unseen data. Validation results form the basis for selecting the best model for testing purpose. Testing data is used to predict the target, which results serve as the true forecast capability of the whole process of training and selecting the best model for making the predictions. The requirement is that there is no look-ahead bias, which means the training, validating and testing exercise is using only the knowledge available until the time of the exercise. The dataset arrangement and the process flow used in Empirical study is illustrated in Figure 10.

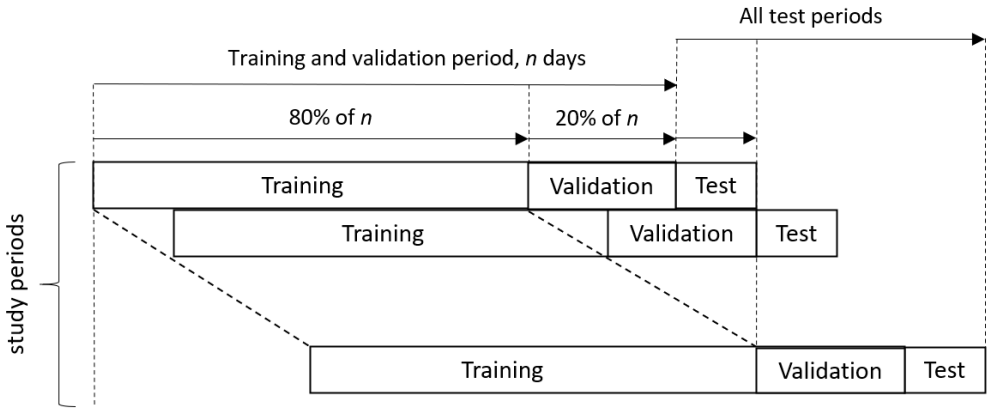


Figure 10. Training and testing arrangement

Next, the model inputs and the target is explained. As we are predicting the stock index return for the next day and neural networks in particular require large datasets, we take stock index logarithmic return for each day, therefore our timestep is one day. We set the macroeconomic and financial markets time-series presented in Appendix A, also arranged in daily timestep for neural network model inputs. The neural network model is fed with input and target and is trained to predict $\hat{y}_{t+1}$ with minimizing the loss in terms of target y_{t+1}.

For the LSTM modeling, it is important that input data is split to rolling windows. In total, there are 45 study periods. Each study period has 780 days and it has been split into datasets of training, validation, and testing, so that each dataset consists of

rolling windows with fixed length o. The number of rolling windows p in each dataset is according to $p = n - (o - 1)$, where n is the number of days in dataset. The study period splitting follows the rule that in testing the prediction is done for 60 days, reserving the last 60 rolling windows for test period. The remaining is split so that training receives 80% and validation receives 20% of data, respectively having 564 rolling windows for training and 141 rolling windows for validation. The day before first prediction in testing period is omitted from validation, reserving it for the situation where time is needed for data to be gathered and trained on models in real life. Rolling window length o is set to 15 days in order to provide sufficient data for network. For each rolling window of features we have assigned target y_{t+1}.

The validation data is merged to the training data in the empirical studies where validation step is not used. The rolling window setting in the Empirical study is illustrated in Figure 11, while actual data is also described in Section 3.2.1.

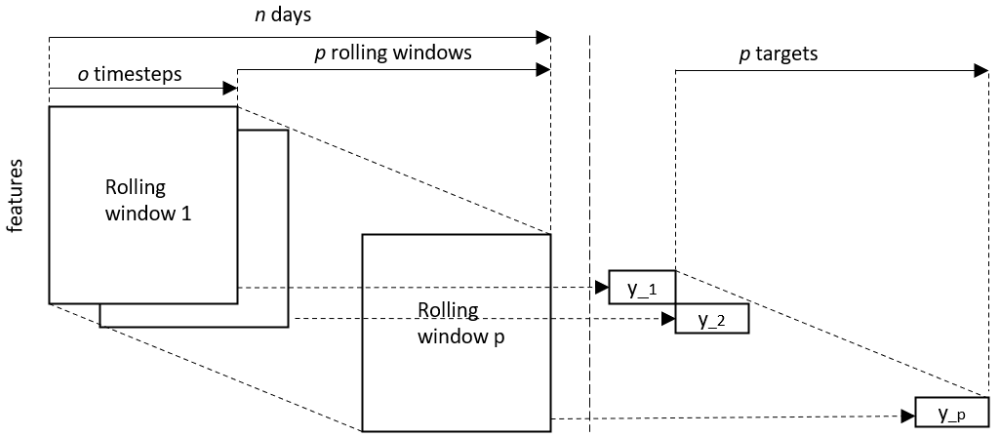


Figure 11. Rolling windows in a dataset

For the prediction, the model receives the time-series of input variables with a number of o observations and predicts only the next day return $\hat{y}_{t+1}$. The predictions are done for one day at a time so that the rolling feature window is updated each time as the window moves forward to next day. Predictions in each test period across study periods are stored sequentially. The resulting prediction array represents all predictions on a given model for the empirical study and it is used for performance measurement.

2.2.2 Data Transformation

Data transformation is often performed in machine learning and neural networks tasks in particular. Data transformation aims to remove, prior to model training, the differ-

ences in data scaling such as different units or scales across different features. Without transformation, the differences in data formats can cause the neural network model to find optimal weights from very different range of values. Data contribution is biased in learning process and can pose difficulties in approximation. The networks trained on standardized data yield better results in general as presented by (M. Shanker, 1996). For these purposes, the relevant data to be used in empirical study is standardized as described below.

Feature y is standardized to $\tilde{y}_i$ by subtracting its sample mean and dividing it by sample standard deviation, as

$$\tilde{y}_i = \frac{y_i - \bar{y}}{s},$$

where $\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$ is sample mean, s is sample standard deviation as

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(y_i - \bar{y})^2},$$

and N is sample size.

Standardization is performed on training data for both features and also for targets. When implementing the model in testing period, we are standardizing testing period input variables using the standardization parameters of training input variables. Because for training data we have prior knowledge of data to be transformed, while testing data remains unknown in terms of future data points, hence also its standardization parameters remain unknown. After standardization, the forecasting during test period outputs predictions in standardized scale. Therefore prediction values are then inverse transformed to get comparable units to true returns and to measure performance.

2.2.3 Methods to improve neural network model performance

In the following section (Goodfellow et al., 2016, Ch.7) is mostly used.

Machine learning algorithm should be designed to perform well not just on the training data, but also on new inputs. Generally, the factors that determine how well the learning algorithm performs this task are the ability to reduce the training error and then reduce the gap between training and test error. This corresponds to the central challenge in machine learning called overfitting, which is when training error has been minimized sufficiently, but test error remains too large. Most known techniques in machine learning for reducing the test error are hyperparameter tuning, regularization and parameter optimization.

The regularization methods impose certain constraints or terms to model parameters with respect to deviation from fixed region or point, which can lead to improved performance

of the test set. In other words, regularization methods aim to increase profitability by reducing the variance without increasing bias too much. While this kind of parameter regularization achieves the purpose by forcing parameters to be close to one another, there might be added advantages to force parameters to be equal. This method of regularization is called parameter sharing. It gives significant advantage because only subset of parameters needs to be stored in memory, leading to great reduction of needed memory in the model. Parameter sharing allows a significantly increase in network size. This is especially important in learning from sequential data such as time series. For that purpose, recurrent neural networks are designed to benefit from parameter sharing, and therefore by default are regularizing parameters, specifically also represented in the LSTM network as the main subject of this thesis.

In regularizing a broad family of models, the dropout method is considered powerful and inexpensive. Dropout trains the network by removing nonoutput units from the network that in the simplest case is done by multiplying the units by zero. For improving the learning process in stochastic gradient descent (SGD), each time a new sample is loaded into a minibatch, there is random sampling of binary mask, which is applied to all the input and hidden units so that the mask is sampled for each unit independently from others. The advantage of dropout is that masking noise is applied to the hidden units, so that it enforces the network to learn from different features and across different units.

Another method called batch normalization is a method of adaptive reparametrization for mostly deep models. When network consists of many layers, then parameter update in each layer is done simultaneously, thereby using updates, that assume other functions in other layers remain constant. Batch normalization can make neural networks faster and more stable through standardization of the input layer parameters, which is similar to feature standardization. As argued in (Santurkar et al., 2019) it also smoothes the objective function that in turn can improve the performance of network model.

Another subset of algorithms to improve training of the neural network and its performance measures is parameter optimization. Once the neural network architecture is set, the optimization aims to configure the neural network model to find the parameters θ of neural network that significantly reduces a cost function $J(\theta)$, to improve performance measure P. Therefore learning process with optimization algorithms act indirectly in terms of objective P. In previous sections we explained gradient descent and it's accelerated version SGD, which is most often used optimization algorithm with its variants. As explained, the crucial hyperparameter for the SGD algorithm is the learning rate. The learning rate setting is considered more of an art than science. It may be chosen by trial and error or by monitoring objective function over time.

The momentum method is an addition to sometimes slow SGD algorithm and is designed to accelerate learning. The momentum algorithm accumulates an exponentially decaying moving average of past gradients. The SGD learning path is corrected with momentum

direction, and the path becomes smoother and faster toward $L_{min}(\hat{y}, y)$. The important variable in momentum is velocity v, which is the direction and speed at which the parameters move through parameter space. Velocity is thought to update the SGD step as the downhill motion affects the particle in physics, pushing the gradient descent to the same direction as in the previous steps, so that the update rule is given by

$$\theta \leftarrow \theta + v.$$

With the Nesterov momentum update, the current velocity is applied before gradient evaluation and velocity update, thus adding a correction factor to the standard method of momentum.

As the learning rate is the most challening hyperparameter to set and largely affects the model performance, and the momentum can mitigate only somewhat the parameter space issues, the alternative learning rate adaptions have been developed. Adaptive learning rates are founded in the idea that if the partial derivative of the loss remains the same sign, then the learning rate should increase, and if it changes sign, the learning rate should decrease. As the empirical study of the thesis also uses some of these adaptive algorithms, the brief explanation to these is given below.

The Adagrad algorithm initially proposed by (John Duchi, 2011) adapts the learning rate of parameters by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient. Thereby parameters that receive high gradients will have their effective learning rate reduced, while parameters that receive small or infrequent updates will increase their effective learning rate. However, the accumulation of squared gradients can result in an excessive decrease of the learning rate.

The RMSProp algorithm (Hinton, 2012) aims to correct premature learning rate decrease by changing the gradient accumulation into an exponentially weighted moving average. Compared to Adagrad, the RMSProp introduces another hyperparameter p, also called decay rate, that controls the length scale of the moving average and unlike Adagrad the RMSProp do not get monotonically smaller.

Adam (Diederik P. Kingma, 2016) is a more recent method and is currently recommended as the default algorithm to use in visual recognition (Fei-Fei Li, 2020). It is also described as RMSProp with momentum, but with the distinction of momentum being an estimate of the first-order moment of the gradient. Adam also includes bias corrections to the both first-order moments and also to the second-order moments to account for their initialization at the origin.

In the empirical study of this thesis, we are at first exercise manual hyperparameter tuning due to computational power limitations and then explore automatic hyperparameter search. Manual hyperparameter search aims is to find the lowest test error subject to

computational power and runtime. This is done by adjusting the capacity of the model to match the complexity of the given task. For example, the more hidden units per layer in the model, the higher is model capacity to represent different and more complex functions. At the same time, too high capacity can lead to the network memorizing the training data and overfit as a result. The optimal model capacity should be targeted somewhere in between. In the empirical study section also different number of epochs are tested.

2.3 Performance Evaluation

2.3.1 Performance Evaluation of Forecasting Model

In earlier section 1.2.5 most relevant loss functions were presented, which can be used to measure and present the machine learning model's performance. The typical regression metrics are the mean absolute error (MAE), and the root mean squared error (RMSE). In presentation of empirical study results, we are using RMSE. For the classification metrics we are using accuracy, which measures how often the predicted label over the training or testing data is accurate in terms of true label (Collet, 2020) or also referred to as a fraction of forecasts that provide the correct sign (Krause and Fairbank, 2020). Accuracy is also motivated by Empirical benchmark method, one of the benchmark methods explained in previous subsection. Accuracy relates to the concept of confusion matrix, as explained in further section 2.3.2. In addition to accuracy, we also measure positive predictive value (PPV) and negative prediction value (NPV). PPV is the fraction of correctly forecasted positive values in the total forecasted positive values. NPV is the fraction of correctly forecasted negative values in the total forecasted negative values. The measures of RMSE, Accuracy, PPV and NPV are used in tables of Empirical Study section of this thesis.

2.3.2 Performance Evaluation of Trading Strategy

In the previous section the machine learning model performance metrics were discussed. For the purpose of trading performance evaluation, we need to introduce a performance concept that takes model predictions and translates that into investment results compared to investment benchmark. Therefore, we need to propose a trading strategy before implementing the forecasting model on the financial instrument. The trading strategy should be tailored to use the forecasting model as profitable as possible. It involves decisions about the timing and scaling of the buy and sell trades. Our forecasting and trading model benchmark is the buy-and-hold strategy, which is buying a certain unit amount of target (total return incl. dividends reinvested) index at the start of the test period and holding it without any further trading until the end of the test period. The primary purpose of neural network models and its implementation via trading strategy is

to provide better returns than the benchmark. In addition, we are interested to find out if LSTM based strategies outperform its benchmark methods. For trading and measuring model performance, we need to construct a trading strategy that would include certain assumptions to compare with the buy-and-hold strategy.

We need to assume that trading strategy utilizes equally the accuracy of both positive and negative predictions. It means trading strategy should have rules that add value over buy-and-hold strategy when predicting both positive and negative returns correctly. For this purpose, we illustrate it with the concept of confusion matrix, which measures the model accuracy in classification tasks. We classify the index returns for each day into positive and negative returns. With that in mind, we seek a trading strategy that gives equal performance impact to both positive and negative return predictions. It can be accomplished by establishing symmetrical trade size around the trading model value, expressed as proportion of current trading model value and named as active weight w_a. The first day of the trading model is starting when we have the first next day return prediction. If the model predicts positive return for the next day, then we invest $(1 + w_a)$ times into the index at today's index closing price, and receive the $(1 + w_a)$ times of the index return by the close price of the index. If there are consecutive positive return predictions, the position in the index is maintained by not doing additional buy trade. If the model predicts negative return for the next day, then position in the index is sold at todays close price at amount that establishes the position in index that delivers $(1 - w_a)$ times the index return for the trade model at next day close price. At the end, the trading model receives on daily basis either $(1 + w_a)$ or $(1 - w_a)$ of the index return. To illustrate the equal performance contribution from w_a under positive and negative returns, we assume there are equal number of observations in each of the four outcomes in the confusion matrix in Table 1.

Table 1. Confusion matrix with equal distribution of outcomes

	Predicted positive returns, $\hat{y}_i > 0$	**Predicted negative returns,** $\hat{y}_i < 0$
Actual positive returns, $y_i > 0$	True Positives, Outperformance $= w_a \cdot y_i$	False Negatives, Underperformance $= -w_a \cdot y_i$
Actual negative returns, $y_i < 0$	False Positives, Underperformance $= w_a \cdot -y_i$	True Negatives, Outperformance $= -w_a \cdot -y_i$

The outperformance generated by trading model in Table 1 is positive performance relative to buy-and-hold index return y, and underperformance is negative performance relative to index return y.

We can see that symmetrical trading sizing allows equal contribution of all four outcomes

in the confusion matrix. Often the trading rules are illustrated as being invested 1 unit of index when positive return is predicted and holding cash as 0 unit invested when negative return is predicted, as in (Faber, 2013). However, this kind of trading rule does not improve the performance of the model in terms of a buy-and-hold strategy in case of true positive prediction, while having model outperformance being dependent only by true negative predictions.

The over- and underweighting concept is often used in investment portfolio management (Lussier and Reinganum, 2020), when managing the sizing of the active positions and measuring the active risks and performance of the model or portfolio against the benchmark. However, the over-and underweight position sizing principle requires simplified assumptions. At first, if after positive return prediction we seek $(1+w_a)$ times overweight in index, it requires w_a times more capital than seeking buy-and-hold return. Similarly, maintaining the underweight position in the trading model requires w_a times less capital, whereby free capital can be invested elsewhere. Assuming that true returns are equally distributed between positive and negative returns, the average capital employed during the whole test period in the trading model should be equal to the buy-and-hold strategy. Therefore, we assume the trading model during the test period does not require on average more capital than the buy-and-hold strategy.

Nevertheless, when initiating an overweight position in the trading model, it requires borrowing w_a times the index market value for the duration of the overweight position. We assume the needed funds can be borrowed from the hypothetical broker at the borrowing rate r_b. Similarly, when initiating an underweight position, we can invest or deposit the available funds w_a times market value of index, with the broker at rate r_i. We make simplistic assumption that $r_b = r_i$, therefore we can accept there is no impact to trading model performance during test period from borrowing and investing the funding of over-and underweight position. Therefore we can further simplify that $r_b = r_i = 0$. It is also supported assumption in case the trading model is used as part of the multi-strategy portfolio in which it gets allocation among other strategies, whereby the over-or underweight of target index is gained by adjusting the weight of this particular strategy in the total portfolio of multi-strategies, when needed.

For next, we attempt to assign optimal w_a for the trading strategy. If daily equity market returns follow random walk, then chances for the next day return being positive or negative are similar to fair coin flip. It has implications to the sizing of our over-and underweight positions as follows. When backtesting the trading strategies, the minimum absolute risk portfolio is set at 0% weight in risky asset, which we also use as a minimum risk portfolio. In our test strategy it would mean 0% weight in the market index. As explained earlier, we prefer symmetrical trading position, then maximum risk portfolio should be 200% weight in risky asset, setting the symmetrical and maximum $\pm 100\%$

active weight deviation in the market index for the trading strategy.

When assigning the optimal over- and underweight position, we use the concept of indifference return within the constant relative risk aversion. Risk aversion means that compensation is required for taking a risk. For risk-averse investors, if the investor puts more wealth at risk, then the marginal utility of taking a risk decreases, because the desire to take additional risk is diminishing. The relative risk aversion (RRA) is a rate at which marginal utility decreases when wealth at risk is increased by one percent. The RRA is used as p in power utility function, as in (Haghani and Morton, 2016) expressing the investor utility as:

$$u(w_a) = \frac{w_a^{1-p} - 1}{(1-p)}.$$

The indifference curve for relative risk aversion connects the size of the wealth w_a, put at risk when trading, to the compensation required for taking that risk. As our range of wealth put at risk is from 0% to 100%, we need to find the wealth in that range at which we are indifferent in taking either 0% weight or 100% weight, given we accept $p = 2$, as suggested by (Haghani and Morton, 2016). Then it means we are indifferent when having 20% wealth at risk with 1% return or 40% wealth at risk with 4% return. As we increase fraction of wealth at risk by f, we require compensation to increase by f^p. Continuing our example, having 100% wealth at risk would require 25% compensation, which is the expected return of such asset itself. The resulting line of Required Compensation across fraction of wealth, presented as x-axis in Figure 12, serves as a cost for indifference curve. The return part of the indifference curve is a fraction of wealth multiplied by rate of asset expected return, presented as Expected Return in Figure 12. An indifference curve is created by plotting the difference of Expected Return and Required Compensation in any given weight, within our minimum and maximum weight range. We can see that the indifference curve is highest at the point of 50% weight at risk. The investor with the abovementioned preferences would not accept more wealth at risk than 50%.

Therefore we assign $w_a = \pm 50\%$ active weight to the market index for our trading strategy. As explained earlier in this subsection, if the model predicts positive return, the index is bought to receive 1.5 times of index return for portfolio. If the model predicts negative return, the index is sold to receive 0.5 times of index return for the portfolio. For example, if the previous recommendation of the model was to be invested in 1.5 times, and the model new forecasted return for next day is a negative value, then index is sold so that to receive 0.5 times the index return for the remaining of portfolio on next day. If the model again forecasts positive return in the following day, then index is bought to receive 1.5 times of the index return for the portfolio going forward.

In the Empirical study, the trading model future value $FV(\hat{\mathbf{y}}, \mathbf{y})$ is measured as compound growth from present value 1, also as multiple of the initial investment at the end of testing

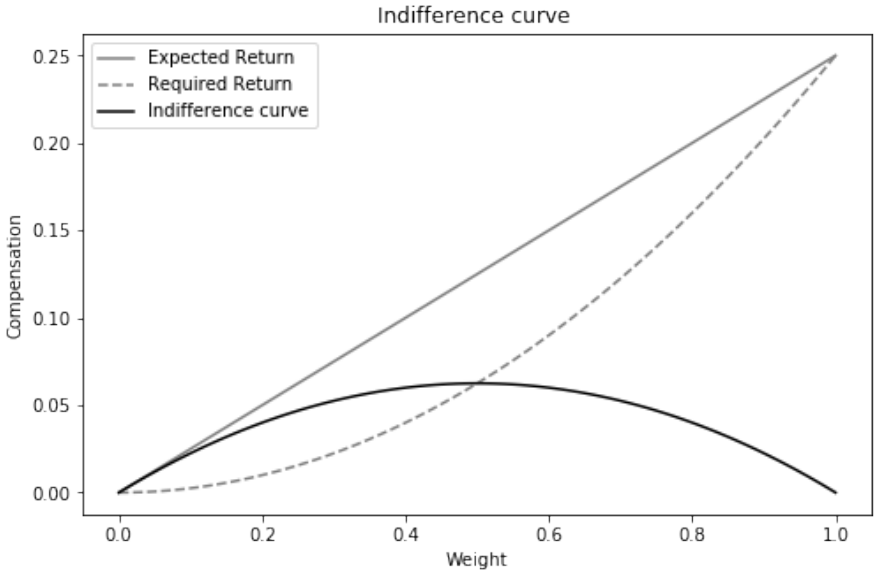


Figure 12. Optimal weight is at a maximum of indifference curve

and is calculated as

$$FV(\hat{\mathbf{y}}, \mathbf{y}) = \prod_{i=1}^{n} \begin{cases} (1 + y_i(1 + w_a)) & \text{if } \hat{y}_i > 0 \\ (1 + y_i(1 - w_a)) & \text{if } \hat{y}_i < 0. \end{cases}$$

The index 'buy and hold' future value $FV(\mathbf{y})$ can be expressed as

$$FV(\mathbf{y}) = \prod_{i=1}^{n} (1 + y_i).$$

As we are interested in model performance relative to index performance, we express such relative trading profit R as

$$R = FV(\hat{\mathbf{y}}, \mathbf{y}) - FV(\mathbf{y}).$$

Relative trading profit R is measure presented in the tables of the Empirical study section of this thesis.

Trading model performance $FV(\hat{\mathbf{y}}, \mathbf{y})$ assumes the previous simplification that $r_b = r_i = 0$, and therefore there are no additional negative returns from borrowing during overweight and no positive returns from investing during underweight index positions. The calculation formula above does not incorporate hypothetical trading costs. In the Empirical study subsection 3.4.1 the total trading cost c are measured to be 33.75% over

the full test period followed by conclusion if neural network trading strategy is profitable or not, after total trading costs c, measured as Relative Net Profit, R_{net} as

$$R_{net} = R - c.$$

3 Empirical Study

3.1 Previous Empirical LSTM research in Financial Markets

The initial motivation of the thesis is founded by (Fischer and Krauss, 2017). Their out-of-sample test results in buying and selling S&P500 index stocks demonstrate the LSTM network is outperforming the standard deep neural networks, random forests and logistic regression. They used one-day returns of the stocks as single feature variable to predict the stock's next day probability of outperformance and underperformance. They conclude LSTM is suitable for financial time-series predictions. The findings also show that the LSTM model has been learning to select stocks mostly based on their characteristic of sudden decrease or increase in the stock price. Interestingly, the conclusion admits this market inefficiency is being recognized in the market and taken advantage of, which explains the decreasing profitability of the LSTM model and its dominant characteristic in their more recent out-of-sample history. The author of this thesis also concludes that neural network might need more diverse set of feature variables to increase the network capacity to approximate true returns, because inefficiencies based on target variable own price history have been arbitraged away by market participants.

(Skabar, 2002) were testing if neural network generated returns on Dow Jones Industrial Average Index are significant by comparing its returns with returns on random walk data, which were derived from the same series by bootstrapping procedure. The findings indicate that stock index time-series are not entirely random and neural networks might achieve better returns than the buy-and-hold strategy. (Bao et al., 2017) proposed deep learning framework where wavelet transforms, stacked autoencoders, and LSTM are combined for forecasting and trading several major stock indexes. Their model testing results show improved returns compared to simple LSTM and RNN. (Chalvatzis, 2019) apply simple LSTM on prediction of major US stock indices, with the added use of predicted return's position within its distribution for improving the profitability of the trading strategy.

3.2 Design of the Empirical Study

3.2.1 Dataset Description

The forecasting task of the model is to predict the OMX Stockholm 30 index returns and to trade based on the model prediction. The OMX index and related dataset is chosen as it represents the liquid stock market, for which the exposure for frequent trading can be gained at a reasonable cost. At the same time it is regional index, with some inefficiences being more likely present than in global stock market indices. Since several related regional financial time series have been available such as currency and interest rates, it allows diversifying the variety of features. It would allow for neural network to discover

and approximate the inefficiencies and relationships among features.

The OMX Stockholm 30 Index (OMX Index) is the stock market index for the Stockholm Stock Exchange. The index consists of the 30 most traded stocks in the exchange, and the index price change measures the weighted capitalization performance of those 30 stocks. The selected features are described in following three groups, presented in Appendix A. Market indicators represent the variants of OMX Index price, such as Open, Close, High, and Low values, and several general financial market indicators on a global and local scale, such as USD/SEK exchange rate, relevant interest rates and others. As shwon in Appendix A, the index closing price s_t is also one of the input variables. The second group is selecting technical analysis indicators, that reflect the OMX Index price strength and the representation breadth of the price strength among OMX Index constituents. The third group of features are fundamental indicators of the OMX Index, such as dividend yield and expected earnings, and few macroeconomic indices representing global and local macroeconomic performance.

For next, the actual data is described, while the data structure for the Empirical study is in Section 2.2.1. As our target is the next day return of the OMX index, the features have daily frequencies. The Economic Leading Index is an exception, as the value is updated on a weekly basis, and therefore, in the dataset its latest values are populated to following days until to new weekly datapoint. All other features have daily frequency in this study. The high frequency of data is important in forecasting with neural networks, because a large number of data is needed for the network to approximate the true returns. The dataset has daily observations from 02.01.2006 to 15.08.2019 and sourced from Bloomberg. The data structure is explained previously in Section 2.2.1.

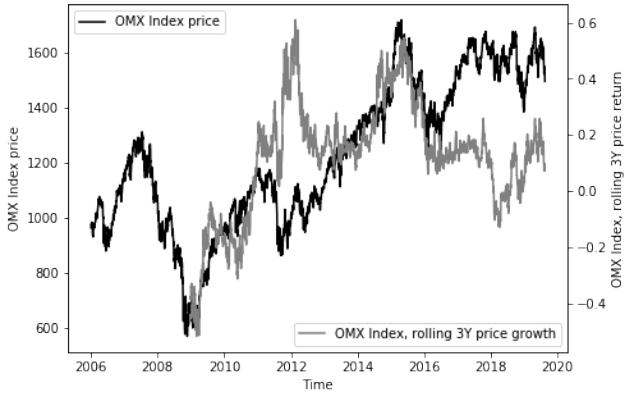
Altogether, 60 days of predictions from each of the 45 study periods are sequentially stacked. In total, it represents 2700 days of continuous out-of-sample predictions, which are being fed into trading strategy implementation. The out-of-sample predictions and trading period is from 11.11.2008 to 15.08.2019.

The characteristics of OMX index data during the entire dataset is presented in Table 2 and Figure 13. The historical data includes global financial crisis during 2008-2009 and commodities crisis which induced global economic slowdown during 2015-2016, therefore presenting stock market behaviour both in expansions and contractions. It is visible in Figure 13 that even if OMX price has increased during the full data sample, it had few severe drawdowns that lasted for many months. There are periods when the market is trending higher or lower, but all the trends have broken eventually. Since the OMX index value has increased from beginning to the end, the mean daily change is positive, and there are slightly more days with positive index change than with negative change. However, compared to mean change, the standard deviation is large, implying the error in terms of any linear approach that forecasts the change based on mean change is challenging. The skewness is very slightly to the left-tail. Even if the data is almost

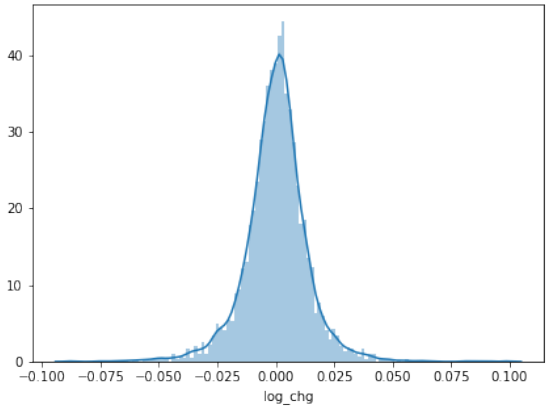
Table 2. OMX index characteristics

Characteristics, daily	**Value**
Mean change	0.0129%
Standard deviation	1.3737%
Number of positive changes	1788
Number of negative changes	1631
Percentage of positive changes	52.30%
Number of direction changes	1786
Skewness	-0.0350
Kurtosis	5.0173
Buy-and-hold in test, $FV(\mathbf{y})$	195.95%

symmetrically distributed by measure of skew, then large kurtosis reveals more negative return outliers and less positive outliers than normal distribution. As it seems OMX index data is non-linear and non-stationary over the full period and it should represent challenging setting for forecasting system, it is interesting to study if the LSTM neural network can forecast based on the data.



(a) OMX index performance



(b) OMX return distribution

Figure 13. OMX Index performance

3.2.2 Software and Hardware

The programming is done in Jypiter Notebook 6.1. The data processing is handled in Python 3.7 with packages *numpy* and *pandas*. The neural network modeling is developed with Keras on top of TensorFlow. Keras *LSTM* layer is used to build the neural network model. Keras *compile* method is used to configure the model for training, allowing the selection of the model parameters such as loss functions and optimizers. Keras *fit* method is used to train the model on data, and then *predict* method is used to forecast. Keras *Tuner* library is used for automatic hyperparameter search. In addition, *Xgboost* package has been used for decision tree gradient boosting modeling. Package *pmdarima* and *statsmodels arima* is used to work with ARIMA models. Python package *sklearn* is used to get values for confusion matrix, which is used for accuracy calculations. The machine learning models are trained on NVIDIA GeForce GTX 1660 Ti.

3.3 Comparison of LSTM Network with Benchmark Models

The first empirical test measures the performance and trading results of LSTM and selected benchmark models. In this section, for any given LSTM model, we use unchanged network architecture and parameter settings across all of the study periods. The LSTM models have been fitted to the same training data. The benchmark models receive the same data as LSTM models, except validation period results are not calculated and validation data is merged to training data. Testing period is the same for all LSTM and benchmark models, so that to present fair comparative evaluation.

Xgboost model uses regressor with default parameter settings. For ARIMA, the *auto.arima* function has been used on training data, to find the ARIMA model during each study period with the lowest AIC score. The resulting best ARIMA models have been used to forecast next-day return during the test period. The testing input data includes all training data for test period and adds new data consequently in each passing day. The ARIMA model is not refitted in the testing period, and only new input data is updated to the model. The training data maintains the quality of using only prior knowledge.

LSTM models have been tested with different values for selected parameters. The LSTM model loss function is MSE and Xgboost uses RMSE for evaluation. LSTM network model consists of a single LSTM layer with tangent activation function set at fixed seed for each experiment. The input sample length $o = 15$ and fitting is performed with standard settings except those described in the results table, which are presented in Table 3. The first part of the tests in Table 3 used only 1 feature being the market index one day return y_t, so that model aims to predict the market return based on the market index returns in the past. In the second part of the tests, the respective models use the full list of 19 features. The improvements in performance metrics of the validation period are observed to discuss which neural network setting might be used in test periods. The test period results are used to assess the whole exercise of training, validating and selecting different LSTM models. The validation results presented in Table 3 have been measured during last the 60 days of the validation period in each study period, in order not to overlap the measurement and be comparable with test period, which has 60 days of trading in each study period.

For the tests in Table 3, the running time to train the model, predict and present results across all the study periods was less than 5 minutes for the lowest capacity models, while the model with the largest capacity completed the process after approximately 8 hours. The model with the best trading performance completed the procedure in less than 20 minutes.

The Naïve model sets the benchmark in terms of RMSE results, illustrating the volatility of the daily returns. The accuracy below 50% means that the daily returns during test period change the return sign in the following day more often than keep the same sign.

Table 3. LSTM and baseline model results

Model	No.of features	Neurons	Epochs	Optimizer	Test RMSE	Test Accuracy	Test Relative profit, R	Validation RMSE	Validation Accuracy	Validation Relative profit, R
Naïve	1			-	0.01847	48.19%	-81.52%			
Empirical	1			-	0.01279	51.30%	-48.08%			
ARIMA	1			-	0.01310	50.70%	63.86%			
Xgboost	1			-	0.01391	50.81%	-35.20%			
LSTM	1	25	50	none	0.01479	49.56%	-15.08%	0.01574	48.96%	-14.87%
LSTM	1	25	100	none	0.01566	50.63%	15.63%	0.01661	50.96%	56.18%
LSTM	1	25	200	none	0.01578	51.33%	200.00%	0.01684	50.15%	118.36%
LSTM	1	25	200	Adam	0.01579	51.33%	153.04%	0.01689	51.00%	67.65%
LSTM	1	50	200	none	0.01563	50.89%	18.19%	0.01659	50.85%	-6.27%
LSTM	1	50	200	Adam	0.01542	51.04%	210.36%	0.01640	51.81%	155.24%
LSTM	1	100	200	none	0.01473	50.81%	38.92%	0.01576	50.93%	14.87%
LSTM	1	100	200	Adam	0.01476	52.15%	285.17%	0.01580	51.33%	185.78%
Xgboost	19			-	0.01600	51.93%	73.82%			
LSTM	19	25	200	none	0.01729	51.30%	164.64%	0.01802	50.67%	23.63%
LSTM	19	25	200	Adam	0.01744	50.30%	69.68%	0.01830	50.22%	-3.96%
LSTM	19	50	200	none	0.01663	50.81%	87.23%	0.01802	49.85%	33.30%
LSTM	19	50	200	Adam	0.01737	49.04%	8.04%	0.01779	50.33%	116.91%
LSTM	19	100	200	none	0.01565	49.26%	38.52%	0.01659	48.63%	-18.62%
LSTM	19	100	200	Adam	0.01574	51.70%	335.00%	0.01669	51.04%	148.30%
LSTM	19	100	2000	Adam	0.01593	52.00%	227.07%	0.01653	50.96%	63.14%
avg LSTM					0.01591	50.81%	122.43%	0.01684	50.51%	62.64%

The Relative trading profit is negative for the Naïve model.

The Empirical benchmark method is the second baseline model, resulting in the lowest RMSE among all baseline models. The accuracy is above 50%, and profit outperforms the Naïve model, but is however negative. The Empirical model results describe characteristics of our data set in terms of accuracy. As explained in the previous subsection of the Empirical benchmark method, the forecasted return for the test period is the mean return from the respective training period in the given study period. As presented in Figure 14 below, such forecasted daily returns in training data generally have positive values since 33 out of 45 study periods have positive daily mean returns.

Empirical benchmark method accuracy at 51.30% suggests that when forecasting the price trend from mostly positively biased training data, the test period has more true positive returns than negative. Empirical benchmark results set the baseline for the accuracy and should compare neural network accuracy against its results, not against 50.00% accuracy. However, the empirical benchmark method failed to take advantage of a relatively accurate price trend forecast. It has no success in trading, resulting in negative trading profits compared to buy and hold strategy. The Empirical benchmark method results are discussed in detail in subsection 3.3.4.

ARIMA is the third baseline model in the test period, and it has used a single feature for fitting. The ARIMA has resulted with the lowest RMSE after the Empirical benchmark, has accuracy at 50.70%, which is close to the average of the LSTM accuracies and

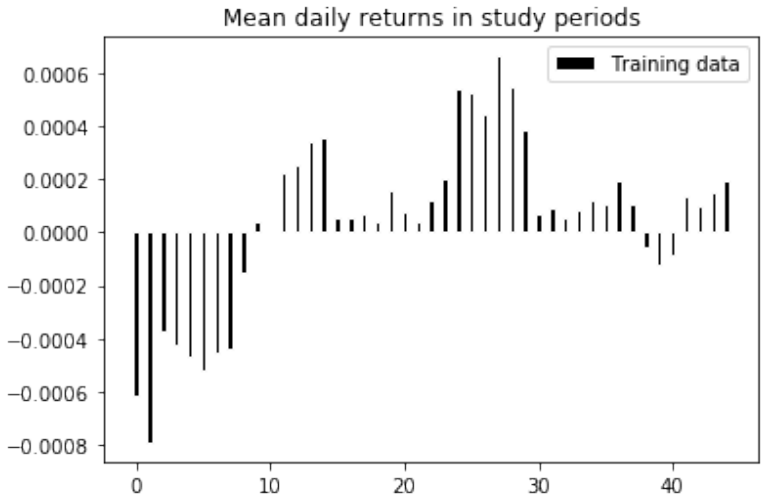


Figure 14. Stock index has mostly positive mean returns in training data

relative profit is positive and the best among benchmark models with a single feature.

The Xgboost is the fourth baseline model, and its RMSE is lower than for the Naïve, but higher than Empirical benchmark and ARIMA. Accuracy is close to the average of all tests with one feature, while profit is negative. The Xgboost performance after using all features resulted in improved accuracy and earned relative profits. This implies Xgboost model did approximate better with a variety of features being used.

3.3.1 LSTM modeling results comparison

For next a series of LSTM predictions have been carried out with variations in some basic parameters such as the number of neurons, number of epochs and the testing with and without using the optimizer. At first, we observe the performance in the testing period. The experiments start with the parameters that give relatively low capacity for the model, and in subsequent tests the capacity has been increased by increasing the parameter values. The results are presented in Table 3 above.

Regarding tests with a single feature, we can conclude the LSTM has on average higher relative profits in testing than baseline models, while in terms of test accuracy the Empirical benchmark prevails. The results of LSTM models with different parameter settings demonstrate that when using low number of neurons (25), the results can be improved by running more epochs, but the addition of optimizer (Adam) does not improve the results. However, when using higher number of neurons and model achieves higher capacity, then Adam helps to prevent the overfitting and gets the best results in

terms of accuracy and trading profits.

For next, the data with 19 features have been used in the fitting and predicting in Xgboost and LSTM models. We are interested if additional internal and external time series data for the stock market index can improve the overall prediction results compared to 1 feature. Again the lower capacity with 25 neurons has been the starting point. The test accuracy in general is lower for given settings compared to experiments with 1 feature. Regarding test relative profits, we cannot conclude that results are better than with one feature. Although when using Adam with 100 neurons, we have achieved the best relative trading profit among all tests in Table 3. The setting with the highest relative trading profits in test period is repeated by increasing number of epochs from 200 to 2000. It has improved the accuracy from 51.70% to 52.00%, while trading profit has declined. It shows that adding more resources for the network has improved the result in terms of learning objective, but not in terms of trading objective.

Therefore we can conclude, that additional features have improved the trading profit only for the parameter setting that gives the highest model capacity in the given set of tests, while accuracy metrics remain somewhat inferior to LSTM models with single feature. Regarding parameter settings, we can observe that the optimizer is not recommended when using lower capacity models, but the test should be run on more epochs. When using higher capacity models, the optimizer is recommended and that is the setting that also resulted in best trading profit result across all tests.

In terms of accuracy metrics, we can observe that higher accuracy is contributing to better trading profits, which is illustrated in Figure 15. Across all the tests performed in this section, the correlation coefficient of Accuracy and Profits is 0.7530.

In terms of the objective function and its metric RMSE, the lower figures achieved in tests have not very obvious relationship with trading profits, as presented in Figure 16. Across all the tests performed in this section, the correlation coefficient of RMSE and profits is -0.2541%, which suggests a stronger relationship between accuracy and profits than RMSE and profits. Therefore, accuracy should be preferred measure over RMSE in neural network forecasting and trading systems.

Compared to baseline models, the best-performing settings of the LSTM network have better RMSE than Naïve but worse than Empirical benchmark, Xgboost, and ARIMA. In terms of accuracy, even if Empirical, Xgboost and ARIMA methods show above 50% accuracy, then best performing LSTM networks have achieved higher accuracy. Across relative profitability, the best performing LSTM settings outperform all the baseline methods. Finally, even if RMSE and accuracy results do not demonstrate LSTM outperformance then in terms of model approximation capability, the LSTM model has been accurate in forecasts when it is most important, as outperforming the baseline methods in terms of profits.

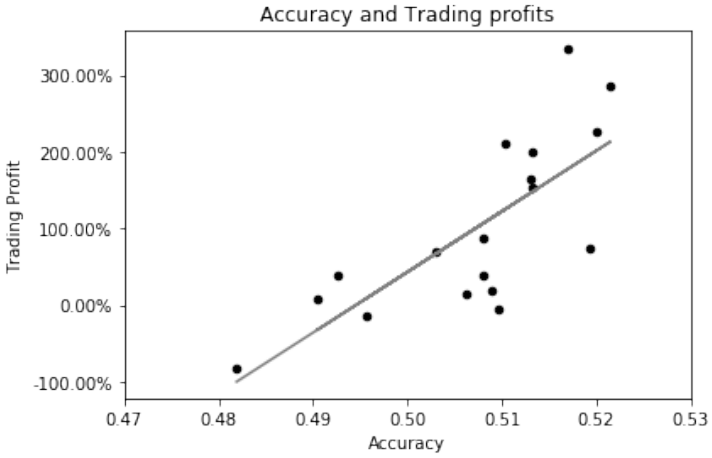


Figure 15. Positively correlated accuracy and trading profits. Test accuracy and trading profit data from Table 3 .

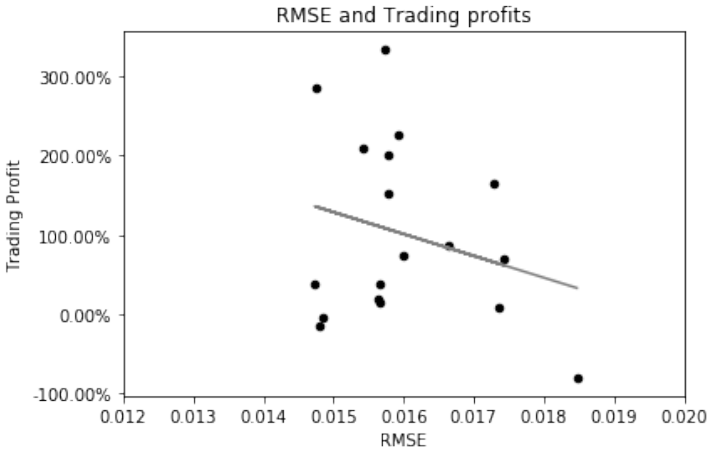


Figure 16. Negatively correlated RMSE and trading profits. Test RMSE and trading profit data from Table 3.

3.3.2 Selection of best LSTM model from Validations for Testing

Next, we select the best performing LSTM model from Table 3 based on validation period result, and check if the selected model maintains good performance during testing. Among performance metrics, we can choose RMSE, accuracy, or relative profits.

The lowest RMSE 0.01574 is for the LSTM model with one feature, 25 neurons, 50 epochs, and no optimizer. However, the model has below 50% accuracy and profits

both in validation and in the test period. Therefore we disregard RMSE as being good predictor of future performance of the LSTM model.

The highest accuracy, 51.81%, during the validation period is from the LSTM model with one feature, 50 neurons, and optimizer Adam. The progress of loss reduction of this model during training and validation periods over epochs is presented in Appendix B. It has accuracy at 51.04% during test peiord, which is higher than average in test period and fourth best relative profit at 210.36%. The next best accuracy 51.33%, in validation is for the LSTM model with one feature, 100 neurons, and optimizer Adam. During test period, it has the highest accuracy 52.15%, and second best relative profit 285.17%. The third best accuracy 51.04%, in validation is for the LSTM model with 19 features, 100 neurons, and optimizer Adam. During the testing period, this model has above average accuracy and the highest relative profit at 335.00%. In summary, out of the three most accurate models in validation, the two have produced the most profitable results in testing period, while third is the fourth most profitable. The LSTM models seem to maintain a similar predictive ability from validation to the test period. We can conclude, that accuracy during validation is a good predictor of performance in test period.

3.3.3 Source of Profitability learned by LSTM model

For next we explore if there is a common pattern in the true index returns, that LSTM model is using to predict. We take the LSTM model Table 3 with the best accuracy during the validation period as an example. We conduct this analysis by collecting all rolling windows of true returns, that have been used as input for the LSTM model predictions in test periods. Since we have 45 study periods, where each test period has 60 predictions daily basis, we have in total 2700 test predictions. For each of the predictions, the LSTM model takes a vector of daily true returns for the last 15 days preceding the prediction day. We collect these returns into 2700 x 15 matrix and then separate them into two matrices. The first has only the return streams, which preceded the positive return predictions, and the second has only the return streams, which preceded the negative return predictions. Then we find the average of the returns for positive return predictions and respectively for negative return predictions for each day in the return stream preceding the prediction.

The resulting averages are presented in Figure 17 and Figure 18, where in x-axis is the vector of days before the prediction, with timesteps from left to right, meaning the rightmost is the last day before the prediction. The y-axis means the input standardized values, which as explained earlier in subsection 2.2.2 are calculated from true returns. We can observe that the LSTM model forecasts positive returns if there has been on average a negative deviation in the last day before prediction, and conversely forecasts negative returns when there has been on average a positive positive deviation in the previous day. The prediction by LSTM seems to exploit the pattern in returns that

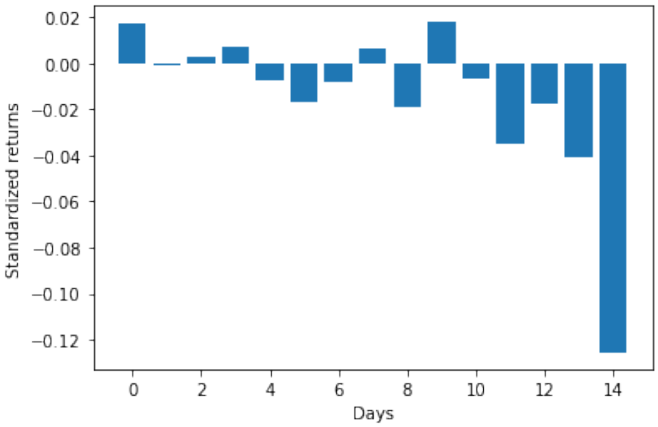


Figure 17. Average input values when LSTM predicts positive returns

reminds of regressing the return prediction on past 1 or 2 values of return. Suppose the prediction value is a regression of it's previous value. In that case, it is called to have some auto-regressive lag order p. Therefore we test the ARIMA models on true index returns with the auto-regressive lag order of 1 or 2, which would attempt to harvest similar characteristics in input.

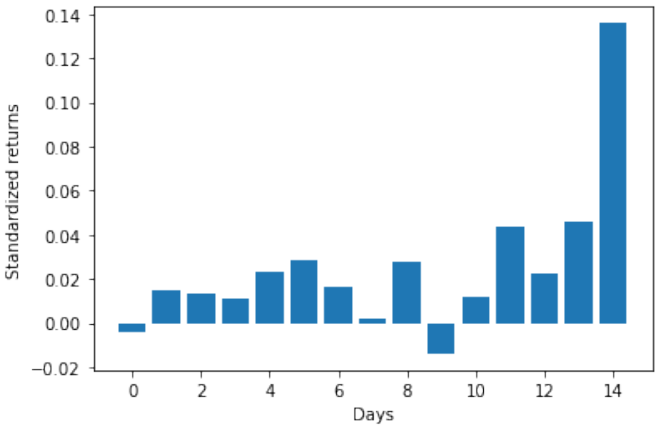


Figure 18. Average input values when LSTM predicts negative returns

If the ARIMA is resulting in trading performance similar to the LSTM model, then it might indeed be the case. As we set p to be either 1 or 2, the exercise has been carried out with all possible ARIMA model combinations where the differencing order d is set to 0 and moving average order q is set to have values either 0 or 1. Differencing order

is 0 as the index return series is considered stationary. The empirical exercise has been carried following the procedure as in Table 3 for ARIMA, except training periods do not have special ARIMA orders. ARIMA(p,d,q) model is fitted for each training period, and then used for the respective test period predictions. Similar to ARIMA in Table 3, the model is not refitted in each timestep during testing, but the new input values are updated to the model. Table 4 presents the best two ARIMA models in terms of accuracy and profitability, where the best performing model is ARIMA(1,0,1).

Table 4. Best LSTM and best ARIMA model results

Model	**No.of features**	**Neurons**	**Epochs**	**Optimizer**	**Test RMSE**	**Test Accuracy**	**Test PPV**	**Test NPV**	**Test Relative profit,** R
LSTM	1	50	200	Adam	0.01542	51.04%	53.60%	48.43%	210.36%
ARIMA(1,0,1)	1			-	0.01279	52.41%	54.58%	49.80%	254.23%
ARIMA(2,0,1)	1			-	0.01280	51.70%	54.00%	49.04%	187.19%

The ARIMA(1,0,1) model has auto-regressive and moving-average error qualities with lag 1. By visually observing Figures 17 and 18, the lag-1 return seems most differentiated datapoint used for prediction by the LSTM model. The profit dynamics of ARIMA(1,0,1) and LSTM over a full test period, is presented in Figure 19.

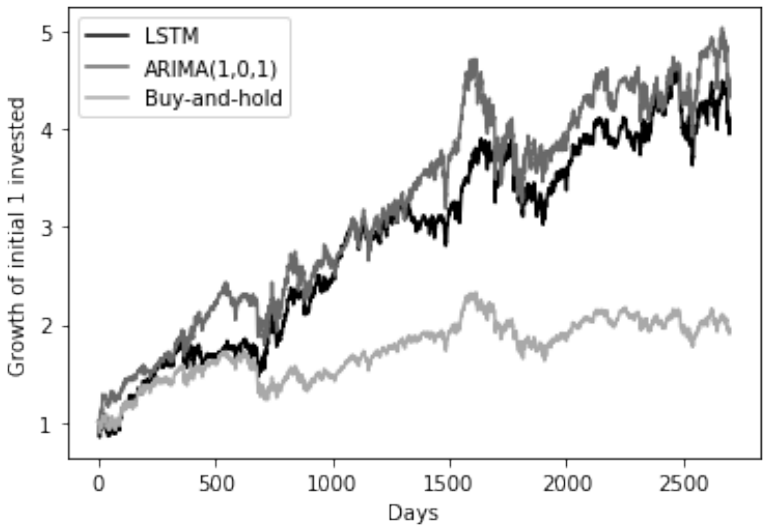


Figure 19. Growth of initial 1 euro invested in LSTM, ARIMA(1,0,1) and in index buy-and-hold, test period

The visual comparison reveals similarities in the profit dynamics. LSTM model has learned to harvest the data characteristics, which also seem to be confirmed in ARIMA(1,0,1) results. In Appendix C, the return predictions in the test period are plotted against their true returns, for LSTM and ARIMA(1,0,1), where their most significant outliers in positive and negative true returns have correct prediction signs quite similar.

3.3.4 Accuracy and Performance Analysis of the LSTM model

As presented in Table 3, the best LSTM model selected from validation has test accuracy similar to the Empirical benchmark, but the difference in relative profits is very large. This subsection aims to find what has caused the most accurate LSTM model selected from validation to trade with such high profits in test period. We try to find the answer by looking if there is particular area of true index returns in test period, where the LSTM model accuracy and performance are better or worse in absolute terms and compared to the Empirical benchmark model. The findings also gave insight if the LSTM model is better suited to rising or declining stock market conditions. The true index returns are the daily returns of the 'buy and hold' strategy presented in the thesis.

The true index returns have been classified into positive daily returns and negative daily returns and further classified into tertiles so that we can assess the results in low, middle and high values for both positive and negative daily returns. The results are presented in Table 5. In addition to accuracy, which is the share of the correctly predicted class, the sum of daily trading returns is presented. Trading returns are calculated according to the trading strategy of the thesis.

Table 5. LSTM and Empirical benchmark accuracy in true return tertiles

True return tertiles	Tertile range	Tertile obs.	LSTM accuracy	Empirical Benchm. accuracy	LSTM sum of returns	Empirical BM sum of returns	True sum of returns
Low positive	0.00%; 0.40%	474	49.36%	79.28%	93.92%	121.66%	94.07%
Mid positive	0.40%; 0.98%	473	50.74%	75.05%	313.26%	387.90%	309.58%
High positive	0.98%; 9.87%	473	53.91%	65.96%	920.54%	960.98%	858.57%
Avg. positive			51.33%	73.38%			
High negative	-8.90%; -1.00%	427	53.86%	31.38%	-788.97%	-941.13%	-809.33%
Mid negative	-1.00%; -0.40%	426	50.00%	22.48%	-280.85%	-357.44%	-281.93%
Low negative	-0.40%; -0.00%	427	48.24%	26.46%	-88.15%	-106.20%	-85.96%
Avg. negative			50.70%	26.80%			

Comparing the accuracies in Table 5, we can observe that LSTM accuracy is above 50% for all Mid and High tertiles, while Empirical benchmark accuracy is above 50% only in positive return tertiles. Therefore LSTM approximation of true returns seems to be

focused on return areas that are crucial the most. LSTM model has learned to be slightly more accurate in positive class, but the difference in accuracy is not meaningful. LSTM has made most of the relative profit against true index returns in the High positive tertile, while relative profits are smaller in negative tertiles.

Another observation is that there is slight imbalance towards positive (52.6%) returns against negative (47.4%) returns in true test returns, based on data in column 'Tertile obs.'. The same calculation is done for predicted returns, and the imbalance is even less, with 1360 positive predictions (50.37%) out of 2700 testing days.

Based on the above, it can be concluded there is no class imbalances in LSTM model predictions. At the same time, this is clearly present in Empirical benchmark model, which has lead the latter to underperform in terms of relative profits. LSTM model forecasts high positive and high negative returns equally well, being equally suitable for both rising and declining market situation.

3.4 Testing Loss Functions

In this section, we present the test results and conclusion of testing the loss functions, which have been described in the thesis. We are using the LSTM parameter setting, which resulted in the best performance in the previous subsection - single layer LSTM network with 100 neurons and optimizer Adam with learning rate 0.001 trained on 200 epochs. In the last section we have been using loss function MSE, and we use it as baseline. We are testing if any loss functions yield better test results than MSE and therefore be considered alternative to MSE. The results are presented in Table 6. The running time for the procedure in each experiment to train, predict and present the results across all study periods was less than 20 minutes, with similar execution times for all the experiments.

The test with mean absolute error (MAE) results in lower test RMSE than with MSE. But the test accuracy 49.41% and relative profit 23.44%) is lower than in MSE test.

The Huber loss test is run on different parameter α values. We can notice that accuracy is above 50% for most of the α values, with below 50% accuracy present in lower α values. The α value 0.1 is the one that achieved higher accuracy than MSE. But we can also observe that using different α values around the best α value 0.1 is yielding quite different accuracy results. None of the Huber test have achieved higher profit than MSE.

Therefore we can conclude that results of Huber loss function are inferior to MSE. While Huber loss results are sensitive to additional parameter α values, the additional complexity has not been rewarded.

The stock loss function is also run on different parameter α values. Similar to Huber loss we can observe that accuracy is above 50% for most of the α values and relative profit is

Table 6. Comparison of loss functions

Loss function	**Test RMSE**	**Accuracy**	**PPV**	**NPV**	**Relative Profit**	**Days Traded**	**Relative Net Profit,** R_{net}
MSE	0.01541	51.04%	53.60%	48.43%	210.38%	1143	153.23%
MAE	0.01512	49.41%	51.95%	46.73%	23.44%	1151	-34.11%
Huber($\alpha = 10.00$)	0.01545	50.74%	53.25%	48.1%	173.83%	1141	116.78%
Huber($\alpha = 2.00$)	0.01550	50.52%	53.10%	47.92%	96.52%	1131	39.97%
Huber($\alpha = 1.00$)	0.01566	50.48%	53.17%	47.95%	4.41%	1121	-51.94%
Huber($\alpha = 0.25$)	0.01542	50.07%	52.71%	47.52%	58.10%	1189	-1,35%
Huber($\alpha = 0.15$)	0.01539	50.26%	52.88%	47.69%	87.08%	1211	26.53%
Huber($\alpha = 0.10$)	0.01517	51.15%	53.82%	48.58%	134.97%	1168	76.57%
Huber($\alpha = 0.05$)	0.01519	49.70%	52.40%	47.23%	6.66%	1187	-52.69%
Huber($\alpha = 0.02$)	0.01505	49.67%	52.33%	47.16%	120.82%	1191	61.27%
Stock_loss($\alpha = 3.00$)	0.01535	50.89%	53.51%	48.31%	105.37%	1183	46.22%
Stock_loss($\alpha = 2.50$)	0.01543	51.96%	54.58%	49.37%	236.57%	1175	177.82%
Stock_loss($\alpha = 2.00$)	0.01557	50.19%	52.82%	47.63%	68.68%	1225	7.43%
Stock_loss($\alpha = 1.50$)	0.01542	49.56%	52.14%	46.94%	69.42%	1205	9.17
Stock_loss($\alpha = 0.50$)	0.01512	52.33%	55.06%	49.75%	485.52%	1169	427.07%
Stock_loss($\alpha = 0.25$)	0.01551	50.52%	53.11%	47.92%	121.57%	1191	62.02%
average	0.01536	50.53%	53.15%	47.95%	125.21%	1174	66.50%

positive for all α values. The best resulting test with α values 0.5 has accuracy 52.33% and relative trading profit 485.52%. The second best α value 2.5 has also outperformed MSE in accuracy and trading performance.

Figure 20 presents the trading performance of MSE, MAE, and best performing Huber and stock loss variants being tested in this subsection. The upper panel presents absolute trading performances and buy-and-hold performance of OMX Index. The lower panel shows the performance relative to buy-and-hold.

We can conclude loss function experiments, that despite of attractive theoretical propositions, the Huber loss has not realized its proposed merits in given neural network settings. The Stock loss function has been able to outperform MSE with the α values that are quite different. As we are not sure how to tune the α values, MSE remains the preferred loss function for the thesis.

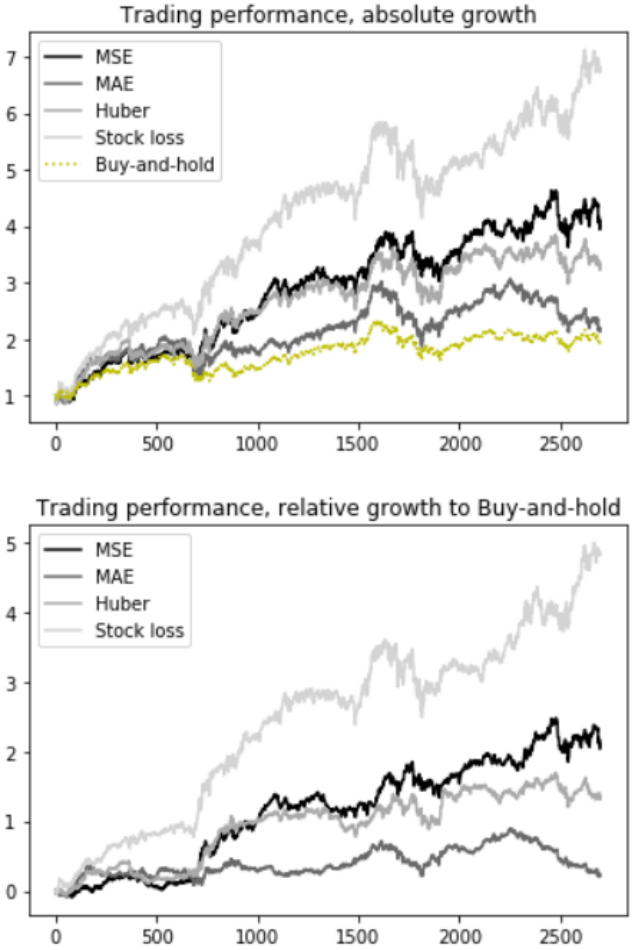


Figure 20. The growth of initial 1 euro invested, using MSE, MAE and most accurate Huber and Stock loss functions

3.4.1 Incorporating trading costs

In this subsection, we discuss the trading costs and measure its effect on profits. We measure the cost of buying and selling the OMX30 index futures as a proxy instrument for gaining the OMX index exposure. As in (Norden, 2009) the average bid-ask spread when trading OMX30 index futures is 0.03%. Assuming the index value is mid-point between the bid and ask, the distance from index value to best bid or ask price is 0.015%. But the actual trade price is often not implemented with the best bid or ask price, and there needs to be cost reserve for slippage. Assuming slippage and broker commissions, we consider the trade cost is 0.05% for either buying or selling to implement the trading strategy. As presented in Table 6, on average across tests with loss functions, there were

1174 days when model traded. As there are 2700 days for trading, this means, on average model traded once in 2,3 days. Based on the 1174 days when traded, the trade cost 0.05%, and the portfolio value traded is 100% at a time, the total trading costs during whole testing period is 58.1%. Therefore, the relative net profit in Table 6, representing the average relative profit net of trading cost, is 66.50% across the test samples of this subsection.

3.4.2 The average capital employed in trading performance

In this subsection, we explore if the trading based on the LSTM model involves the different capital employed during the test period in terms of buy-and-hold. Also, in case the capital employed is more significant, then if this leverage applied on average throughout the test period on true index returns would have resulted in better performance than buy-and-hold performance. In other words, we seek to find out if the LSTM model suggests additional leverage through trading strategy and if this itself applied throughout the test period can generate relative profits. We investigate in the LSTM model experiment with MSE loss function, presented earlier in Table 6 in this section.

Out of 2700 trading days in the test period, the LSTM model predicts 1360 days of positive returns and 1340 days of negative returns. As positive return prediction results in 1.5 times of index return and negative prediction in 0.5 times of index return, respectively applying financial leverage or remaining partly uninvested, then on average during test period the trading portfolio is invested 1.0037 times of the index returns. We assume it is the average leverage ratio during test period, as it describes the ratio of invested total capital to investor own capital. If the leverage ratio is above 1, it means a loan is taken to invest in the index. If the ratio is below 1, it means not all of the own capital is invested in the index, leaving some capital uninvested, consistent with assumptions presented in Section 2.3.2.

When multiplying each true daily return by leverage ratio 1.0037 and measuring the compounded growth of an investment based on this return stream during the test period, we arrive to leverage adjusted buy-and-hold return, which is 96.27%. The buy-and-hold return 95.95%, is presented in Table 2. As leverage adjusted buy-and-hold return almost the same as buy-and-hold return, it means that the average leverage ratio inherent in trading the LSTM model predictions is not meaningful and has no impact to trading performance.

3.5 Automatic Hyperparameters Search

The configuration options of the neural network that influence how the parameters will be learned are called hyperparameters (Versloot, 2020). These are typically optimizer and

learning rate, batch size, number of epochs, and number of neurons. In the previous sections, we selected hyperparameter configurations manually. Also, in previous subsections, the tests have been carried out with the unchanged LSTM network architecture and fixed hyperparameter values across all study periods. It means as we tested different values of neurons, epochs, and the value addition of optimizer, these values were fixed across all our study periods. This subsection aims is to search the best hyperparameters for each study period during training and then implement these hyperpameter values in test data of the respective study periods. For that purpose we use different automatic hyperparameter search methods available in the Keras Tuner library version 1.0.1 (O'Malley et al., 2019), which was able to exercise with Tensorflow 2.1. in Python 3.6.1.

Keras Tuner allows multidimensional search space, but we are using one- and two-dimensional search spaces due to our computational constraints. At first, we search separately across a different number of neurons, then at different learning rates, and finally combined learning rate and number of neurons space. Among other search methods we are implementing three search strategies - Random Search, Hyperband, and Bayesian Optimization. Random Search tries a random combination of hyperparameters from a given search space (Conol, 2020). HyperBand (Li et al., 2018) is based on an algorithm that provides speeding up random search through adaptive resource allocation and early-stopping. Bayesian Optimization represents Gaussian process that maps the hyperparameters to probabilities on objective function, while evaluating probabilities based on past evaluation results, which then gives guidance for selecting the next configuration to evaluate, as in (Conol, 2020).

Hyperparameter search ranges have been set as neurons: $[minvalue = 25, maxvalue = 150, step = 25]$ and learning rate =$[0.01, 0.001, 0.0001, 0.00001]$. As we perform hyperparameter search, Keras Tuner requires setting the maximum number of trials within the search space and the number of executions of each trial, which we have set at 10 and 3 respectively. Further, Adam is selected for optimizer, the training and validation split is set at 80% and 20%, search objective is for Validation Loss and tuner search is set to perform 100 epochs. Keras method for fitting training data is using the same configuration as for the best-resulting configuration in the previous subsection: training and validation split set at 80% and 20%, batch size 32 and 200 epochs. The hyperparameter search results are presented in Table 7, with search methods, testing RMSE, accuracy, and profitability figures for the full length of testing periods. The running time for the procedure to search hyperparameters and then train the models, to predict and present the results across all study periods took less than 2 hours for one dimensional search. In comparison, for two dimensional it took less than 4 hours.

Table 7 demonstrates that among one-dimensional search spaces, the searching across learning rates has resulted in lower RMSE and higher accuracy and profit figures than searching across a number of neurons. Another observation is that compared to one-

Table 7. Automatic hyperparameter search results

Hyperparameter space	Search Model	RMSE	Accuracy	PPV	NPV	Relative Profit, R
neurons	RandomSearch	0.016350	49.52%	52.38%	47.31%	94.67%
neurons	Hyperband	0.016793	48.26%	50.94%	46.22%	-13.69%
neurons	BayesianOptimization	0.016354	49.26%	52.04%	47.03%	36.64%
learning rate	RandomSearch	0.015512	51.30%	54.52%	48.90%	83.77%
learning rate	Hyperband	0.015370	50.26%	52.83%	47.71%	118.09%
learning rate	BayesianOptimization	0.015818	50.19%	53.11%	47.88%	76.61%
neurons, learning rate	RandomSearch	0.015168	50.11%	52.62%	47.51%	93.74%
neurons, learning rate	Hyperband	0.015740	51.48%	54.11%	48.94%	191.62%
neurons, learning rate	BayesianOptimization	0.015247	50.89%	53.76%	48.46%	110.42%

dimensional search spaces, the two-dimensional search space has resulted better accuracy and profit with Hyperband and BayesianOptimization, while RandomSearch accuracy is giving mixed results. From a theoretical point of view, the Random Search has perhaps the weakest proposition among other search models especially as due to computational constraints the maximum number of trials is set to low number, as is the case here. Therefore we can conclude, that generally multi-dimensional search space should be preferred over one-dimensional search space.

In terms of assessing overall results of Keras Tuner search methods compared to results without Keras Tuner and fixed hyperparameters across study periods, we can observe that results using hyperparameter space are inferior. For the RMSE, accuracy, and profit figures, none of the search models and search spaces outperformed the best fixed hyperparameter configuration (neurons 100, learning rate 0.001, 200 epochs). But generally using search methods should from conceptual point of view make the model more robust, since it is testing more configurations and is less prone to possible bias emerging due to using fixed hyperparameters across different study periods.

3.6 Effect of Randomness on Empirical test results

The focus of this subsection is to assess the importance of initial weights in the neural network model, which are assigned randomly. During neural network initialization, the initial weights of the nodes are created randomly by programming language. For reproducibility of the conducted experiments and the comparability of the experiments across the thesis, the randomness has been so far fixed in all the experiments in the thesis. It is done by fixing the same seed value before the neural network model is built and fitted to the data, and it ensures that network is using the same initial weights across different experiments. In this subsection we explore if different random seeds may produce better or worse results in terms of performance metrics, how much the results are deviating in

our sample tests, and finally assessing the likelihood that neural network can be expected to provide accuracy above 50% and positive relative profits over different random seeds.

For that purpose, we are taking the neural network configuration, which had the best experiment results in terms accuracy and profits, as in Table 8. Then the network is rerun several times using different randomness (seed value) each time and results are showed in Table 8, sorted by the accuracy from highest to lowest. Table 8 presents the average and spread between the highest and lowest performance metrics values. The running time for the procedure to train the model and forecast the output across all study periods was less than 20 minutes for each experiment.

Table 8. Results from set of randomly initialized weights

Random Seed Value	**RMSE**	**Accuracy**	**PPV**	**NPV**	**Relative Profit,** R
4	0.015220	52.70%	55.30%	50.00%	237.35%
6	0.015363	52.63%	55.24%	50.04%	190.03%
17	0.015405	52.63%	55.31%	50.04%	246.89%
5	0.015180	52.44%	55.14%	50.00%	378.27%
16	0.015205	51.19%	53.82%	48.61%	176.66%
13	0.015328	51.15%	53.92%	48.62%	172.10%
3	0.015420	51.04%	53.60%	48.43%	210.38%
18	0.015349	51.00%	53.68%	48.45%	220.15%
9	0.015500	50.70%	53.28%	48.10%	110.52%
8	0.015655	50.44%	52.99%	47.82%	42.28%
12	0.015419	50.44%	53.14%	47.92%	266.44%
2	0.01535	50.33%	53.11%	47.87%	154.82%
14	0.015429	50.19%	52.82%	47.63%	114.58%
20	0.015562	50.15%	52.72%	47.54%	75.09%
19	0.015426	50.00%	52.60%	47.42%	69.96%
1	0.015660	49.96%	52.53%	47.35%	67.90%
7	0.015326	49.85%	52.42%	47.23%	165.11%
10	0.015599	49.81%	52.43%	47.25%	102.58%
11	0.015695	49.67%	52.19%	46.98%	-42.80%
15	0.015677	49.59%	52.26%	47.09%	50.08%
average	0.015440	50.80%	53.43%	48.22%	150.42%
max - min	0.000519	3.11%	3.12%	3.13%	421.07%
sample st.dev.		1.04%			

In Table 8, 15 out of 20 tests have resulted in accuracy above 50%, with the mean accuracy 50.80%, which reasonably assures the network can provide above 50% accuracy. The relative profit is positive for 19 out of 20 tests and the average profit at 150.42% is attractive. But the results demonstrate that initial weights given to network can have a large impact on the performance metrics and further research is needed to find initial network parameters that might perform well consistently.

3.7 Discussion and Future Work

Primary motive for the research in the thesis has been to explore if additional economic features can improve the performance of the neural network and trading, compared to single feature OMX index price return. However, in the given dataset and tested network settings, the results demonstrate that adding a wide range of features did improve the accuracy of LSTM model. Potential future work might discover, if any feature selection methods can reduce the dimensionality and improve the results in the given dataset. Combined with testing at different lengths of rolling timestep windows, it could help select the features and historical data structure best accustomed to the modeling purpose.

Further research can also be done on network architecture, like deep neural networks and multi-layer LSTMs, which have not been tested in the thesis. The model tuning can be further experimented by adding the other methods not tested but described in the thesis, such as dropout, batch normalization, or alternative optimizers. The thesis discovered that LSTM learned simple price pattern in the given data. A deeper network and adding selectively feature could be explored, to assess if and which patterns in data could be approximated by the LSTM model.

Regarding empirical results, the LSTM network outperformed the benchmark methods described in the dataset in terms of relative profits. When discussing the accuracy and trading performance, the trend in target variable time-series is an important factor. The accuracy of neural network model should be compared to the accuracy of the empirical benchmark model, which considers the trend in training data. Across the different network configurations, the resulting accuracies did not exceed on average the Empirical benchmark method. But perhaps one of the significant observations from tests is that neural network has been able to be accurate in times of most importance - when true index returns are in medium and higher absolute values. The latter quality has led to the trading profitability of LSTM models being superior to empirical benchmark methods.

The thesis results show that higher capacity models tend to perform better, especially if combined with the usage of optimizer. Even if higher capacity models, in theory, are prone to overfit, the results here demonstrated the opposite. Since the dataset arrangement is 45 non-overlapping test periods with 60 days of investing, it ensures that results were not affected by chance. If the computational resource is an issue and does not allow for high capacity models, then lower capacity models could also be used without optimizer.

When considering using either accuracy or RMSE for model selection, accuracy is showing stronger relationship with profits than RMSE, and accuracy is preferred metric for model selection.

As class imbalance turned to be relatively low for the best LSTM model in validation, a trading strategy could be explored further to utilize the better accuracy in higher absolute

values of true returns. Different thresholds and hurdles can be investigated for trading, for more efficient harvesting of higher accuracy in larger absolute values of true returns.

The stock loss function returned the highest single test profit among all the tests in thesis and the future work could investigate if there is a method to find the optimal α parameter, which could enhance the profits. Until then, MSE should be preferred when working with LSTM and time series. Huber function has attractive reasoning however, the MSE results were superior.

Automatic hyperparameter search is a conceptually attractive addition to neural network modeling. In terms of computational resources it is the most demanding addition, but if resources allow it should be further explored. In the thesis, the resources enabled to build and test only minimal search configurations, such as two-dimensional search space with a small number of trials. But still, the results were superior to benchmark methods and were attractive in terms of trading performance. Results of automatic hyperparameter search are conceptually similar to *auto.arima*, and the latter could serve as a good baseline procedure.

Trained models were affected by the random initialization of network weights. Across different random initial weights, the models on average performed well. The testing should be done on several weight initializations to mitigate the risk that the weights are randomly either beneficial or harmful for performance metrics.

4 Conclusions

The first objective of this thesis has been to present fundamental concepts of artificial neural networks while focusing on applying the LSTM model on the OMX stock market index forecasting. The appropriate data arrangement, forecasting, and performance measurement methodology have been described. Several methods to improve the neural network performance have been discussed as well as the performance metrics of the network model.

The second objective of the thesis has been to assess the LSTM model performance against benchmark models. The accuracy is the preferred measure over RMSE in selecting the best models, as accuracy is more correlated with higher profits. The results showed the accuracy of the LSTM model is not superior to benchmark models. Still, the LSTM model can be more accurate in terms of higher absolute values of true returns, which have given the edge over the benchmark models and resulted in higher profitability.

The third objective has been to construct a relevant trading strategy for implementing the model predictions. The selected approach suggests staying in 1.5 times leveraged position in the index when positive return is predicted, and conversely to remain in 0.5 times invested in the index when negative return is predicted. Performance results show that trading strategy can harvest LSTM model accuracy, resulting in net of trading costs outperformance against buy-and-hold index.

The results showed that a large feature set did not improve performance results in terms of RMSE, accuracy, and profits. The best performing models used single feature. The tests also revealed that higher capacity models performed better than lower capacity models and optimizer improved performance in higher capacity models.

LSTM model accuracy during the validation period has been used as basis to select the best model for the testing period. The results show that accuracy and profitability achieved in the validation period are persistent in the following test period, suggesting LSTM model and proposed trading strategy can generate profits in similar timeframe as used in the thesis.

The return data used to predict with the LSTM models reveals the pattern of declining return before the positive return predictions and conversely rising return before negative return prediction. The ARIMA model that approximates similar input data characteristics, has been tested. The profit profile of ARIMA and LSTM shows similar dynamics, which indicates both models have utilized same pattern in return data.

Huber loss and Stock loss have both attractive propositions, but did not show consistency in parameter α values, adding the complexity of parameter evaluation. Mean squared

error has been the preferred function across the empirical studies in the thesis.

Automatic hyperparameter search was done based on several search models. The relative profits outperformed benchmark models, but underperformed some of the manually set hyperparameter settings. Still, automatic parameter search should provide more reliable setting of hyperparameters, as it allows to tailor the hyperparameters across many study periods, which would not be possible manually.

Finally, the last experiments tested the impact of randomly initiated weights on the performance. It appears the initial weights have significant effect on accuracy and profits. The average accuracy achieved across different initial weights has been very close to the most accurate benchmark models. Neural network models should aim for measures to mitigate the effect of initial weights to model performance.

References

Bao, W., Yue, J., and Rao, Y. (2017). A deep learning framework for financial time series using stacked autoencoders and long-short term memory. *PLoS ONE*, 12(7):e0180944. https://doi.org/10.1371/journal.pone.0180944.

Chalvatzis, Chariton; Hristu-Varsakelis, D. (2019). High-performance stock index trading: making effective use of a deep long short-term memory network. *University of Macedonia, Department of Applied Informatics*. https://arxiv.org/pdf/1902.03125.pdf.

Collet, F. (2020). Introduction to keras for researchers. Retrieved on May 14, 2021, from https://keras.io/getting_started/intro_to_keras_for_researchers/.

Conol, C. (2020). Hyperparameter tuning with keras tuner. Retrieved on May 14, 2021 from https://towardsdatascience.com/hyperparameter-tuning-with-keras-tuner-283474fbfbe.

Davidson-Pilon, C. (2016). Bayesian methods for hackers: Probabilistic programming and bayesian inference. Retrieved on May 14, 2021 from https://camdavidsonpilon.github.io/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/.

Diederik P. Kingma, J. B. (2016). Adam: A method for stochastic optimization. *3rd International Conference for Learning Representations*. https://arxiv.org/abs/1412.6980.

Faber, M. (2013). A quantitative approach to tactical asset allocation. *The Journal of Wealth Management*. https://ssrn.com/abstract=962461.

Fei-Fei Li, Ranjay Krishna, D. X. (2020). *CS231n: Convolutional Neural Networks for Visual Recognition*. Stanford University. Retrieved on May 14, 2021 from http://cs231n.stanford.edu/.

Fischer, T. and Krauss, C. (2017). Deep learning with long short-term memory networks for financial market predictions. *FAU Discussion Papers in Economics*, (11). https://www.econstor.eu/handle/10419/157808.

Flach, P. (2017). *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 11th edition.

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

Haghani, V. and Morton, A. (2016). Optimal trade sizing in a game with favourable odds: The stock market. https://ssrn.com/abstract=2875682.

Haykin, S. S. (2009). *Neural networks and learning machines*. Pearson Education, Upper Saddle River, NJ, third edition.

Hinton, G. (2012). Neural networks for machine learning. University of Toronto. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

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

Hyndman, R. J. and Athanasopoulos, G. (2018). *Forecasting: Principles and Practice*. OTexts: Melbourne, Australia, 2nd edition. https://otexts.com/fpp2/.

John Duchi, Elad Hazan, Y. S. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*. https://jmlr.org/papers/v12/duchi11a.html.

Krause, A. and Fairbank, M. (2020). Baseline win rates for neural-network based trading algorithms. IEEE WCCI 2020 ; Conference date: 19-07-2020.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2018). Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52. http://jmlr.org/papers/v18/16-558.html.

Lussier, J. and Reinganum, M. R. (2020). Active equity investing: Portfolio construction. CFA Program Refresher Readings.

M. Shanker, M.Y. Hu, M. H. (1996). Effect of data standardization on neural network training. *Omega The International Journal of Management Science*.

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press. http://neuralnetworksanddeeplearning.com/index.html.

Norden, L. L. (2009). Asymmetric futures price distribution and bid-ask quotes. *Stockholm University School of Business*. http://ssrn.com/abstract=1456846.

Olah, C. (2015). Understanding lstm networks. Retrieved on May 14, 2021 from http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

O'Malley, T., Bursztein, E., Long, J., Chollet, F., Jin, H., Invernizzi, L., et al. (2019). Keras Tuner. Retrieved on May 14, 2021 from https://github.com/keras-team/keras-tuner.

Rosenblat, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

Sam Lau, Joey Gonzalez, D. N. (2019). Principles and techniques of data science. https://www.textbook.ds100.org/about_this_book.html.

Santurkar, S., Tsipras, D., Ilyas, A., and Madry, A. (2019). How does batch normalization help optimization? MIT. https://arxiv.org/pdf/1805.11604.pdf.

Skabar, Andrew; Cloete, I. (2002). Neural networks, financial trading and the efficient markets hypothesis. 25th Australasian Computer Science Conference.

Trevor Hastie, Robert Tibshirani, J. F. (2017). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 12th edition. https://web.stanford.edu/~hastie/ElemStatLearn/.

Versloot, C. (2020). Automating neural network configuration with keras tuner. Retrieved on May 14, 2021 from https://www.machinecurve.com/index.php/2020/06/09/automating-neural-network-configuration-with-keras-tuner/.

Vicente, R. (2019). Neural networks. Course material, LTAT.02.001, Institute of Computer Science, University of Tartu. https://courses.cs.ut.ee/2019/nn/spring/uploads/Main/nn_lecture_8_2019_spring_slides.pdf.

Appendices

A Appendix

Market indicators	**Description**
Open, Close, High, Low	Index price when trading open, close index price highest and lowest values
VIX Index	US equity market volatility index
USD/SEK	USD/SEK exchange rate
STIBOR 3 month	Sweden interbank lending market interest rate
SWE 10y yield	Swedish Government 10 year bond yield
Industrial Metals	Industrial Metals combined price index by Bloomberg
Technical Indicators	
RSI 14D	Relative Strength Index, 14 days
MACD	Difference of moving average convergence divergence (12,26) and signal (9)
Declining volume	Sum of the number of shares traded for the stocks in the index which price was lower
RSI 14D less than 30	Percent of index members with a 14-day RSI value less than 30
MACD above 0	Percent of members with MACD > Baseline 0
Fundamental Indicators	
Dividends	OMX Index sum of gross dividend per share gone ex-dividend over prior 12 months
Estimated earnings	Weighted estimated earnings of index stocks
Estimated dividend yield	Average analysts estimated dividend yield
EPS F12 Est	Average analysts estimated earnings per share 12-month forward
Economic Surprise Index	Citi Economic Surprise Index for Sweden
Economic Leading Index	ECRI US Leading Index, weekly

B Appendix

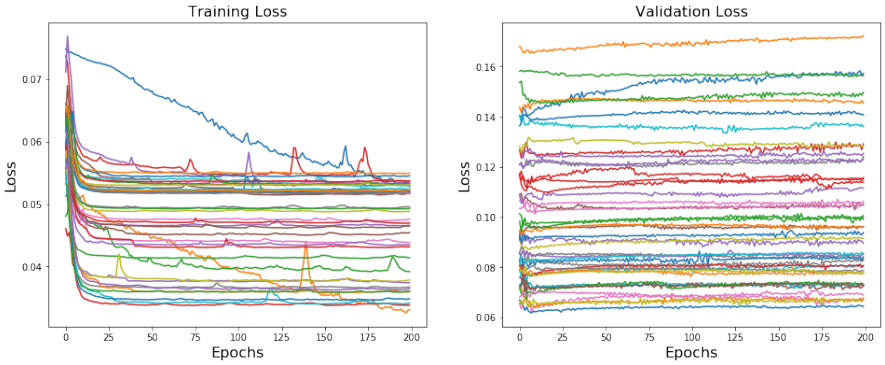


Figure 21. Training and validation loss in each study period, best validation LSTM model

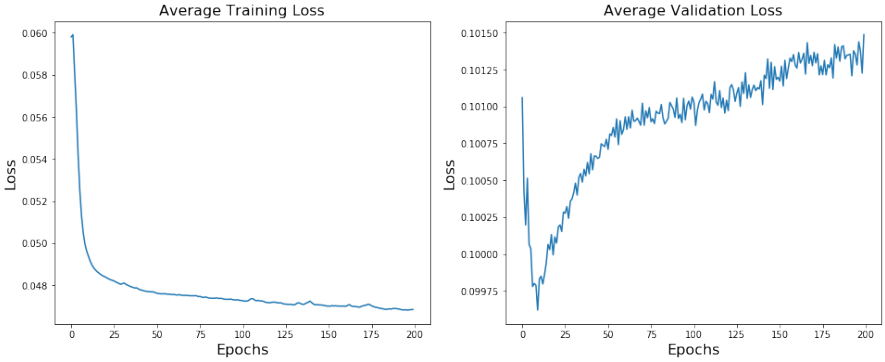


Figure 22. Average training and validation loss across study periods, best validation LSTM model

C Appendix

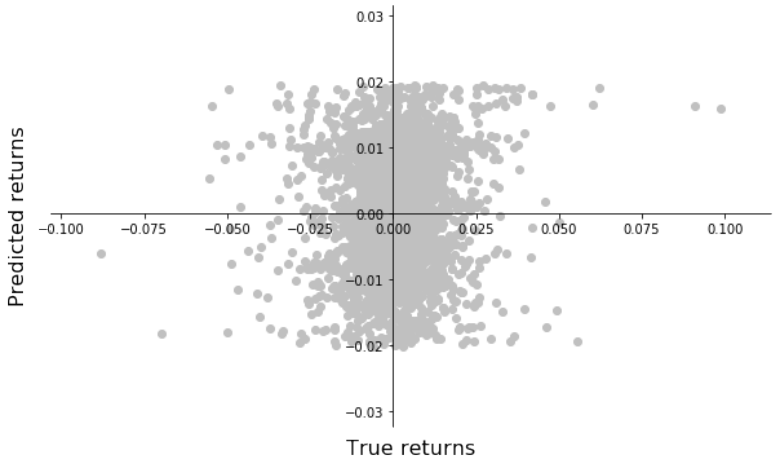


Figure 23. Best LSTM model return predictions in test period and true returns

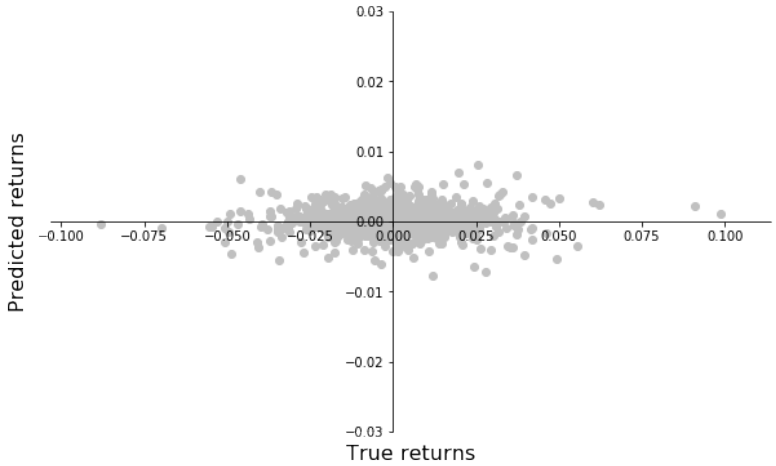


Figure 24. ARIMA(1,0,1) return predictions in test period and true returns

Non-exclusive licence to reproduce thesis and make thesis public

I, **Vahur Madisson**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding DSpace digital archives until the term of the copyright,

   Forecasting and Trading Financial Time Series with LSTM Neural Network, supervised by Toomas Raus and Meelis Kull,

2. I grant the University of Tartu a permit to make the work specified in p.1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Common Licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of the copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other person's intellectual property right arising from the personal data protection legislation.

Vahur Madisson

14.05.2021