CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Real Time Distributed Stock Market Forecasting using Feed-Forward Neural Networks, Market Orders, and Financial indicators

A distributed multi-model approach to stock forecasting

Master's thesis in Computer science and engineering

Oscar Carlsson, Kevin Rudnick

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Real Time Distributed Stock Market Forecasting using Feed-Forward Neural Networks, Market Orders, and Financial indicators

A distributed multi-model approach to stock forecasting

Oscar Carlsson, Kevin Rudnick

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Real Time Distributed Stock Market Forecasting using Feed-Forward Neural Networks, Market Orders, and Financial indicators
A distributed multi-model approach to stock forecasting
Oscar Carlsson, Kevin Rudnick

Supervisor: Philippas, Tsigas
Examiner: Andrei, Sabelfeld

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Real Time Distributed Stock Market Forecasting using Feed-Forward Neural Networks, Market Orders, and Financial indicators
A distributed multi-model approach to stock forecasting
Oscar Carlsson, Kevin Rudnick
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Machine learning and mathematical models are two tools used in prior research of stock predictions. However, the stock market provides enormous data sets, making machine learning an expensive and slow task, and a solution to this is to distribute the computations. The input to the machine learning in this thesis uses market orders, which is a different way to make short-term predictions than previous work. Distributing machine learning in a modular configuration is also implemented in this thesis, showing a new way to combine predictions from multiple models. The models are tested with different parameters, with an input base consisting of a list of the latest market orders for a stock. The distributed system is divided into so-called *node-boxes* and tested based on latency. The distributed system works well and has the potential to be used in large systems. Unfortunately, making predictions with market orders in neural networks does not provide good enough performance to be viable. Using a combination of predictions and financial indicators, however, shows better results.

Keywords: Machine learning, deep neural network, distributed systems, stock market prediction, market orders.

Acknowledgements

We would like to thank our supervisor Philippas Tsigas for the guidance, support, and advice during the thesis.

We would also like to thank our examiner Andrei Sabelfeld for taking the time to read trough our thesis and giving us feedback.

Finally we want thank our partners-in-life Amanda and Sonja for supporting us throughout this thesis and always keeping spirits high. Long live beters.

Oscar Carlsson, Kevin Rudnick, Gothenburg, June 2021

Contents

List of Figures **xiii**

List of Tables **xvii**

1 Introduction **1**
1.1 Aim of The Project . . . 2
1.2 Risk and Ethical Considerations . . . 2
1.3 Limitations . . . 3
1.4 Thesis outline . . . 3

2 Theory **5**
2.1 Stock market momentum . . . 5
2.2 Lagging and leading indicators . . . 5
2.2.1 Exponentially weighted moving average . . . 5
2.2.2 Relative Strength Index . . . 6
2.2.3 Moving Average Convergence Divergence . . . 6
2.2.4 Volatility . . . 7
2.2.5 Price Channels . . . 7
2.3 Machine Learning . . . 7
2.3.1 Neural networks . . . 7
2.3.2 Loss function . . . 8
2.3.2.1 Backpropagation . . . 9
2.3.3 Activation functions . . . 10
2.3.3.1 ReLU . . . 10
2.3.3.2 Leaky ReLU . . . 10
2.4 Normalization . . . 10
2.4.1 Min-max Normalization . . . 11
2.4.2 Z-Score Normalization . . . 11

3 Previous work **13**

4 Methods **15**
4.1 Data . . . 15
4.1.1 Building features . . . 15
4.1.1.1 Price . . . 15
4.1.1.2 Time . . . 15
4.1.1.3 Financial indicators . . . 16

4.1.2 Matching x and y data . . . 16
4.1.3 Graphs . . . 17
4.2 Distributed system . . . 17
4.2.1 Node-Box . . . 17
4.2.2 Smart-sync . . . 18
4.2.3 Applying Techniques in Layer 1 . . . 21
4.2.4 Coordinator . . . 21
4.2.5 Coordinator Protocol . . . 21
4.2.6 Redundancy . . . 23
4.2.7 Parallelization . . . 23
4.2.8 Cycling node-boxes . . . 23
4.2.9 Financial Indicator Node . . . 24
4.2.10 Test Bench . . . 25
4.3 Stock predictor . . . 25
4.3.1 PyTorch . . . 25
4.3.2 Neural networks . . . 26
4.3.3 Input data . . . 26
4.3.4 Output data . . . 27
4.3.5 Training . . . 28
4.3.6 Testing . . . 28

5 Results **31**
5.1 Feed-forward Neural Network . . . 31
5.1.1 Min-max normalization . . . 31
5.1.2 Z-normalization . . . 32
5.1.2.1 Swedbank . . . 32
5.1.2.2 Nordea . . . 33
5.1.3 Discussion . . . 35
5.2 Distributed System . . . 37
5.2.1 Smart-sync . . . 37
5.2.2 Node-Boxes latency . . . 37
5.2.3 Discussion of Smart-sync . . . 39
5.2.4 Discussion of Node-boxes . . . 39
5.3 Distributed combined models . . . 41
5.3.1 Discussion . . . 44
5.4 General remarks . . . 45
5.4.1 Convergence Towards Average . . . 45
5.4.2 Offset in X and Y axis . . . 45
5.4.3 Deep and Shallow Network performance . . . 45
5.4.4 Network instability . . . 46
5.4.5 Financial Indicators Combined With Market Orders . . . 47

6 Conclusion **49**
6.1 Future work . . . 50
6.1.1 Long Short-Term Memory and Transformers . . . 50
6.1.2 Train model with data from several stocks . . . 50
6.1.3 Volume indicators . . . 50

6.1.4 Train using node-boxes 50
6.1.5 Optimize node-box code 51
6.1.6 Classification predictor 51

Bibliography **53**

A Appendix 1 **I**

List of Figures

4.1 The base of a Node-Box 18
4.2 Node-Box with Swebank's 200 latest market orders 18
4.3 Three nodes in layer 1, with their targets connected to one node in layer 2 19
4.4 Basic structure of the smart-sync. 19
4.5 Smart-sync with data added. The data in green is completed data that has been sent to be processed. The data in yellow is still missing data. 20
4.6 The dark green color indicates old data that has been overwritten. . . 21
4.7 Nodes in layer 1 taking turns to calculate their target, possibly achieving higher processing power 21
4.8 Node-Boxes asking the coordinator who and how to communicate . . 22
4.9 Two Node-Boxes are doing the same calculations to achieve redundancy. 23
4.10 Two Node-Boxes doing different calculations to parallel processing. . . 24
4.11 Two nodes cycling their workloads, increasing the processing power . 24
4.12 FI-Node, which gives financial indicators as features to the Node-Box in layer 2 25
4.13 Graph showing how the data is split in the distributed system. Test L1 overlaps Train L2, Eval L2, and Test L2 as the same dataset is used. L*x* means Layer *x*. 29

5.1 Graphs for predictions of three models using min-max normalized data with different window sizes; 70, 200 and 700. All graphs depict the same time period 31
5.2 Two graphs for Swedbank stock price prediction using model 70_100e_lr0.0001_S. Left graph shows prediction for 3 hour window of Swedbank stock. The left shows a 30min window. Larger figures can be found in Appendix I. 33
5.3 Two graphs for price prediction using model 200_50e_lr0.0001_S. Left graph shows prediction for 3 hour window of Swedbank stock. The left shows a 30min window. Larger figures can be found in Appendix I. 34
5.4 Two graphs for price prediction using model 700_100e_lr0.0001_S. Left graph shows prediction for 3 hour window of Swedbank stock. The left shows a 30min window. Larger figures can be found in Appendix I. 34

5.5 Two graphs for price prediction using model *70_100e_lr0.0001_N*. Left graph shows prediction for 3 hour window of Nordea stock. The left shows a 30min window. Larger figures can be found in Appendix I. 35
5.6 Two graphs for price prediction using model *200_100e_lr0.0001_N*. Left graph shows prediction for 3 hour window of Nordea stock. The left shows a 30min window. Larger figures can be found in Appendix I. 36
5.7 Two graphs for price prediction using model *700_30e_lr1e-05_N*. Left graph shows prediction for 3 hour window of Nordea stock. The left shows a 30min window. Larger figures can be found in Appendix I. 36
5.8 Graph showing the difference between Algorithm A and B. 38
5.9 Graph showing the fastest, mean, and slowest time it took to run n node-boxes in layer 1, with one node-box in layer two. The test is done over a 5 minute interval and is measured from the time a node in layer 1 starts its processing, until the node in layer 2 calculates it prediction from that timestamp. 39
5.10 Graph shows stock price predictions for the distributed model *vol50_lr0.01_None_D*. Pred70, Preds200 and Preds700 refers to the prediction inputs used for the distributed model. Predictions for a 3 hour Swedbank window. 42
5.11 Graph shows stock price predictions for the distributed model *vol50_lr0.01_None_D*. Pred70, Preds200 and Preds700 refers to the prediction inputs used for the distributed model. Predictions for a 30 min Swedbank window. 42
5.12 Graph shows stock price predictions for the distributed model *ema15_macd_rsi5_rsi30_vol100_vol50_lr0.001_NordeaPred70_D*. Pred70, Preds200 and Preds700 refers to the prediction inputs used for the distributed model. Predictions for a 3 hour Swedbank window. 43
5.13 Graph shows stock price predictions for the distributed model *ema15_macd_rsi5_rsi30_vol100_vol50_lr0.001_NordeaPred70_D*. Pred70, Preds200 and Preds700 refers to the prediction inputs used for the distributed model. Predictions for a 30 min Swedbank window. 44
5.14 Price prediction result of a section of the test data for Swedbank_A, using model **ema_70_35E_30s_1e-06_time1**. X-axis is numbering of datapoints from the start of the test set. Data collected from 08/25-2020 to 15/3-2021 46
5.15 Price prediction result of a section of the test data for Swedbank_A, using model **price_200_5E_15s_1e-06_time1**. X-axis is numbering of datapoints from the start of the test set. Data collected from 08/25-2020 to 15/3-2021 46
5.16 Average test loss for two independent runs using the same data. Normalization is not applied. 47
5.17 Average test loss for two independent runs using the same data. Normalization applied. 47

A.1 Prediction of Nordea stock with 70 market orders as input, using model *70_100e_lr0.0001_N*. The graph is shown over a 3 hour window. II

A.2 Prediction of Nordea stock with 70 market orders as input, using model *70_100e_lr0.0001_N*. The graph is shown over a 30 minute window. II
A.3 Prediction of Nordea stock with 200 market orders as input, using model *200_100e_lr0.0001_N*. The graph is shown over a 3 hour window. III
A.4 Prediction of Nordea stock with 200 market orders as input, using model *200_100e_lr0.0001_N*. The graph is shown over a 30 minute window. III
A.5 Prediction of Nordea stock with 700 market orders as input, using model *700_30e_lr1e-05_N*. The graph is shown over a 3 hour window. IV
A.6 Prediction of Nordea stock with 700 market orders as input, using model *700_30e_lr1e-05_N*. The graph is shown over a 30 minute window. V
A.7 Prediction of Swedbank stock with 70 market orders as input, using model *70_100e_lr0.0001_S*. The graph is shown over a 3 hour window. VI
A.8 Prediction of Swedbank stock with 70 market orders as input, using model *70_100e_lr0.0001_S*. The graph is shown over a 30 minute window. VII
A.9 Prediction of Swedbank stock with 200 market orders as input, using model *200_50e_lr0.0001_S*. The graph is shown over a 3 hour window.VIII
A.10 Prediction of Swedbank stock with 200 market orders as input, using model *200_50e_lr0.0001_S*. The graph is shown over a 30 minute window. VIII
A.11 Prediction of Swedbank stock with 700 market orders as input, using model *700_100e_lr0.0001_S*. The graph is shown over a 3 hour window. IX
A.12 Prediction of Swedbank stock with 700 market orders as input, using model *700_100e_lr0.0001_S*. The graph is shown over a 30 minute window. IX
A.13 Prediction of Swedbank stock using the distributed model *rsi30_lr0.01_None_D*. The graph is shown over a 3 hour window. . . X
A.14 Prediction of Swedbank stock using the distributed model *rsi30_lr0.01_None_D*. The graph is shown over a 30 minute window. XI

List of Tables

4.1 The hardware specification of the test bench. 25

5.1 MSE and MAE test-set losses for models using min-max normalization. Swedbank between 1 Mars - 19 April 32
5.2 Table showing loss scores over Swedbank test set, 1 Mars to 19 April for three different models. Losses for a 10 minute average strategy and the offset strategy is also shown. Swedbank single models (the best ones, used in dist) 70 uses time, 200, 700 does not (Deep 30s all) 33
5.3 Table showing loss scores over Nordea test set, 1 Mars to 19 April for three different models. Losses for a 10 minute average strategy and the offset strategy is also shown. 35
5.4 Comparison between Algorithm A and B with different input-sizes. . 38
5.5 Node-boxes benchmark as $n : m$, where n is the number of nodes in layer 1, and m the number of nodes in layer 2 38
5.6 Table shows MSE and MAE losses for layer two models used in the distributed network. Sorted by MSE loss. Data is Swedbank stock price for 13 April - 19 April 41
5.7 Table shows MSE and MAE losses for top performing distributed models and single stock models. Data is Swedbank stock price for 13 April - 19 April 41

1
Introduction

The stock market introduces considerable monetary opportunities for investors with their ears to the ground. Buying when the market is low and selling when it is high is a common saying in the stock market community. However, employing this strategy with good timing and precision is a challenging task. To stand a chance, investors use various predictive techniques to gain some market insight. However, with the massive amount of traded stocks each second[47] it can be a daunting task to forecast this massive market. Traditionally investors have used mathematical models for different kinds of technical analyses, such as the Black-Scholes model [24], the Heston model [9] or the Gamma Pricing model [25]. These models are proficient and are still in use today. Other financial tools often used in conjugation with pricing models are financial indicators such as the Moving Average Convergence Divergence (MACD), the Relative Strength Index (RSI), and the Stochastic Oscillator (KDJ). Traditionally an investor needs experience to combine all these models and indicators into a final quality prediction. Studies show that these technical analysis techniques can increase profitability when being acted upon compared to a buy-and-hold strategy [6, 50, 37].

Even though several pricing models have shown promising results, some studies still debate whether the stock market is predictable [44]. The efficient market hypothesis and random walk hypothesis [26] are two examples of such theories debating stock market predictability, both of which state that the stock price is not based on historical value or historical movement but instead based on new information. Moreover, the hypothesis state that the value of a stock is precisely its price and is determined instantly based on new information. An opposite school of thought states that the market is indeed predictable and thus is not arbitrarily random. It states that the market price moves in trends and that market history repeats itself. Since market price is determined (in the end) by humans selling and buying stocks, the human psyche determines the market, which does not necessarily react instantly to new information and is prone to follow trends.
In the hopes of detecting such price trends, several researchers have deployed price prediction models using neural networks [11, 29]. Neural networks are function approximators that can learn advanced patterns in data. Furthermore, neural networks are universal, meaning they can make predictions on any data, thus not limiting choices of input data. For example, Chong et al.[5] created a deep neural network that inputs several 5min intervals of stock return data from several different stocks, showing the potential of cross-stock data. Qian and Rasheed [36], using daily return data, increased their prediction accuracy with an ensemble of several

machine learning models, combining artificial neural networks, decision trees, and k-nearest neighbors. This ensemble outperformed all individual models.
In the stock market, there are two main types of transactions; market orders and limit orders. Market orders are placed at the current market price and executed nearly instantly. The buyer or seller can not decide the price of the transaction. Setting an asking price lower or higher than the market price creates a limit order, which executes when it matches someone else's market order. Several researchers have created models which aim to forecast stock market price based on limit order status [45, 41], and other researchers have created models using fixed interval price data, for example, daily price or 5min interval price segments. However, to the best of our knowledge, no research exists describing the use of market orders directly in real-time.

1.1 Aim of The Project

The main contribution of this thesis will be to create a model that attempts to forecast the stock market using market orders in real-time. Additionally, the massive quantity of stock information, financial indicators, and earnings reports that can be gathered from the stock market indicate the need for a system that can utilize several such inputs at once. Our goal is to create a system that inputs several different types of predictive information and creates a final prediction. We hypothesize that by utilizing several models, prediction prices from different stocks, and various window sizes, the combined prediction could produce a better result than any single model. Additionally, since we wanted to display the computational power potential of a distributed system, using market orders was a natural decision.

1.2 Risk and Ethical Considerations

There are several considerations to have in mind when working with the stock market. First of all, legalities must be taken into considerations, such as *Pump and Dump Schemes[39]* and *Insider Trading[38]*. In the case of this thesis, there is no trading or social interaction done, so there is no risk of any impact on the stock market. As there is no impact on the stock market, there is no need to consider any legalities.

An ethical issue to consider is the idea of machine learning trying to understand human behavior for the possibility of earning money (when someone else will lose it). In this case, the program will try to predict stock prices by learning from its previous history. As it will only look at individual market orders, which can be seen as individual trades between humans, it will solely base its prediction on human behavior. It will try to find patterns such as optimism and pessimism, which could correspond to an upcoming rise or fall in the stock market price. However, this is solely speculation, and the program might find other patterns that a human cannot see or understand.

1.3 Limitations

As no historic market order data was found for Stockholmsbörsen, the data was collected daily as soon as the thesis was proposed. The short time resulted in a limited amount of historic market order data, which could impact the performance of the machine learning algorithms.

Our data will consist of market orders, on which we will perform our forecasting. However, on the stock market, one can perform several different orders. Limit orders are one of those. Limit orders can contain a lot of predictability information, as shown by Zheng[53]. However, we will not use this data and limit ourselves to market orders.

The system will use historical stock data, as previously mentioned. Thus, the system will not use live stock market data. Using live data would have been an intriguing step to take, but it would add little value to the academic goal of this thesis.

The testing of the distributed part of the thesis was done on a single computer, as there was not possible to access an extensive network of computers. It would theoretically work over several computers, but it will not be possible to get any performance statistics from this thesis.

1.4 Thesis outline

Six chapters divide the thesis. After the introduction, the theory chapter explains different machine learning aspects and financial indicators.

In chapter 3, we examine previous work of stock prediction with machine learning and distributed machine learning. Several different techniques that inspired this thesis are mention.

Chapter 4 explains the machine learning components used in our predictive models and presents the distributed system.

Chapter 5 describes the result of the thesis and discusses those results. The main focus of the results is the performance of the predictions, but it also includes some benchmarks of the distributed system.

Chapter 6 is the final chapter which contains the conclusion and future work.

2
Theory

In this chapter, we present relevant background information and theories. The first sections describe the stock market and relevant financial indicators. Following this, machine learning information is explained, such as neural networks, backpropagation, and normalization.

2.1 Stock market momentum

Stock market momentum can be a good indicator for deciding between a long or short position for any investor. Momentum in the stock market serves to measure the overall trend based on historical data. Several researchers have shown that following a strategy based on stock market momentum can be highly profitable [12, 17]. Many indicators can measure stock market momentum, such as RSI or MACD, which are described in further detail below.

2.2 Lagging and leading indicators

A *lagging indicator* is an indicator that aims to explain the past. In doing so, one might find specific trends that indicate what will happen in the future. For example, if the unemployment rate went up last month and this month, one could say that it seemingly will rise again next month. This prediction is, of course, not certain, as the unemployment rate can not climb indefinitely.
A *leading indicator* on the other hand, says what will happen in the future. Events that have happened but have not yet affected the process at hand are the basis for these indicators. For example, if customer satisfaction is way down, the company performance might not be affected yet, but it might be in the future.

2.2.1 Exponentially weighted moving average

Exponentially weighted moving average (EMA) is a technique used to calculate the average of rolling data. What makes EMA unique is that it uses the constant α to determine how much impact older data should have on the new average. The expression used for EMA is as follows:

$$\hat{x}_k = \alpha\hat{x}_{k-1} + (1 - \alpha)x_k$$

where $0 < \alpha < 1$. $\hat{x}_k$ is the calculated EMA, α is the constant to determine the importance of older data, also known as the *filter constant*. $\hat{x}_{k-1}$ is the most recent EMA, and x_k is the current value [43].

2.2.2 Relative Strength Index

Relative Strength Index (RSI) is a financial indicator that shows the momentum of a stock. It is an oscillator that ranges from 0 to 100. It is calculated by comparing the decrease and increase of closing prices over a certain period. Presented by Welles wilder Jr in 1978, this indicator has seen significant use among investors. Traditionally, an RSI value under 30 is considered a buy signal, and a value over 70 a sell signal. On average, the RSI value is 50, meaning any RSI over this value indicates a possibly overbought security, and anything under a possibly oversold security [42].

When calculating the RSI value, a time window first needs to be determined. Welles Wilder presented a period of 14 periods as an appropriate window. The periods could be any time intervals, for example, days, weeks, or months. Then two exponentially moving averages are calculated. One over any periods where the closing price is down, and one for periods with a higher closing price. The RS value can be determined by:

$$RS = \frac{EMA_{UP}}{EMA_{DOWN}}$$

The following formula is used to convert this relative strength value into a value between 0 and 100, :

$$RSI = 100 - \frac{100}{1 + RS}$$

The result RSI is the relative strength index value.

2.2.3 Moving Average Convergence Divergence

Moving Average Convergence Divergence (MACD) is a momentum-based indicator that shows the relationship between long and short-term exponentially moving averages. In other terms, it helps decide if the market is overbought or oversold compared to the expected trend, the long-term *exponentially moving average*. MACD value is calculated simply by subtracting a short-running EMA by a long-term EMA; thus, a negative value indicates that the security is underperforming short term, and vice versa for a positive MACD value [52].
Normally the MACD value is based on a 12 period EMA and a 26 period EMA and can thus be calculated by the following formula:

$$MACD = EMA_{12} - EMA_{26}$$

Any MACD movement that crosses zero typically indicates a buy or sell signal.

2.2.4 Volatility

Volatility is a statistical measure of the dispersion of a stock. Dispersion is the expected range for which a value is predicted to fall within, in other terms, the uncertainty of a particular position. The uncertainty could be determined by, for example, the returns or risk of a portfolio.

If the price of a stock falls and climbs rapidly over a certain period, the security can be considered volatile. How much the price oscillates around the mean price in a time segment can be interpreted as the volatility value. Thus stocks with high volatility have less predictability and are considered higher risk than stocks with low volatility.

The variance of the price over some time defines historical volatility. The standard deviation can measure historical volatility during this period [10].

2.2.5 Price Channels

Price channels are indicators for the highest and lowest price trends over a time segment. Donchian channels are a way to calculate price channels over different time segments. The highest point, not including the current timestamp, of the stock price during a time segment calculates the top channel. During the same time segment, the stock's minimum price calculates the bottom channel. The current price is not included in the time segment to make it possible to see if the current price breaks the current trend of the top or bottom channel. Breaking the current trend could indicate a future bear or bull trend [7, 4].

2.3 Machine Learning

Machine learning is a process in which a computer improves a model's predictability power by finding patterns in data. By processing a large amount of sample data points from some distribution, the computer can learn how to interpret the data and output an accurate result. These inputs could be pre-collected annotated data or data gathered live via interaction with an environment. Machine learning can solve many different problems, including item classification, regression, and text processing. Different machine learning algorithms are thus more suited to specific problems than others. For example, a decision tree will not be as suited for image classification as a Convolutional neural network (CNN) [28].

2.3.1 Neural networks

Neural networks are a type of machine learning built by layers of weights and biases that can learn most classification or regression tasks. Often portrayed as a network of nodes, a neural network can train to approximate any non-linear function. Between nodes, an activation function is used that introduces non-linearity to the model, explained in further detail below. Typically a neural network has several layers, starting with the input layer. This layer inputs some vector x, which then propagates

forward through the network, finally reaching the output layer. Layers between the input and output layers are called hidden layers. The term deep neural networks refer to networks with multiple hidden layers. Neural networks require a larger amount of data than other machine learning approaches since they contain more parameters to optimize than other machine learning techniques.

2.3.2 Loss function

Loss functions, or cost functions, present a way to evaluate the performance of classifiers. It does so by representing the error of some predictions compared to the target with an actual number. Intuitively this is needed since simply measuring a classifier in simple wrong, or correct terms does not provide any numerical scale regarding how accurate the classifier is. For example, if a model can classify the animal species of a picture, such as a cat, predicting a dog is better than predicting a whale.

As predictions improve, the value given by the loss function decreases; thus, training a classifier is an optimization problem where we seek the function f, which maps inputs to outputs that minimize the loss. Let $L(\cdot, \cdot)$, be the loss function and $L(f(x_i), y_i)$ be the loss for prediction $f(x_i)$, where x_i is the input vector, and for y_i the target value. For N training samples the optimization problem is [48]

$$\min_f \frac{1}{N} \sum_{i=1}^{N} L(f(x_i), y_i)$$

Choosing a loss function is an integral part of solving this optimization problem satisfactorily. For example, a typical loss function for classification tasks is the cross-entropy loss. The cross-entropy loss is defined as:

$$L(f(x_i), y_i) = y_i log(\sigma f(x_i))$$

where $\sigma(\cdot)$ is the probability estimate, and y_i is the target label for data point i. The greater probability value from σ for a correct answer will yield a lower loss.

For regression tasks, a typical loss function is the Mean Squared Error (MSE), also known as L2 loss.

$$MSE = \frac{\sum_{i=1}^{n} (y_i - f(x_i))^2}{n}$$

where y_i is the target value for data sample i and $f(x_i)$ is the predicted value [20].

An alternative way to calculate loss is to use the mean absolute error (MAE), calculated as following:

$$MAE = \frac{\sum_{i=1}^{n} |f(x_i) - y_i|}{n}$$

where $f(x_i)$ is the prediction, y_i is the target value for data sample i, and n is the number of values [49].

2.3.2.1 Backpropagation

Backpropagation is a method for training and fitting neural networks. The word *back* refers to the method used in doing this, where gradients are calculated from end to start, backward, in order to calculate them efficiently. Gradients are the direction that a function is increasing. Thus, by calculating all gradients for all weights with respect to a loss function, one can use the gradients with an optimization algorithm, such as Stochastic gradient descent.

Calculating gradients using more traditional approaches is done by calculating the gradient for each weight independently, which grows exponentially in a neural network. On the other hand, backpropagation utilizes previously calculated gradients, finding all gradients in linear time.

If J measures the error between an output from the model and the target, then ΔJ is the movement of the loss function, which we are trying to minimize. We call ΔJ the gradient for the loss function that builds up a vector containing all partial derivatives of weights and biases.

Let, x: the input vector
y: the target vector
J: the loss or error function
L: amount of layers in neural network
W^l: the weights connecting layers $l - 1$ and l
b^l: the bias for layer l
f^l: activation function for layer l

A single layer in the neural network has the following structure:
$z^l = W^l \quad a^{l-1} + b^l$
$a^l = f^l(z^l)$
where a^l is the output of layer l and a^{l-1} is the previous layers output.

The gradients as mentioned before is calculated from end to start, thus using the chain rule:
$\frac{\delta J}{\delta W^l} = \frac{\delta W^l}{\delta z^l} \frac{\delta a^l}{\delta z^l} \frac{\delta J}{\delta a^l}$

This can be expanded to each layer of the network since:
$\frac{\delta J}{\delta a^{l-1}} = \frac{\delta z^l}{\delta a^{l-1}} \frac{\delta a^l}{\delta z^l} \frac{\delta J}{\delta a^l}$
Thus reusing some of the derivatives of layer l for finding the derivatives for layer $l - 1$. This propagates through the nodes until the start is reached in which case $a^0 = x$.

Using all the partial derivatives for the weights and biases, the gradient ΔJ is then:

$$\Delta J = \begin{bmatrix} \frac{\delta J}{\delta W^1} \\ \frac{\delta J}{\delta b^1} \\ \dots \\ \frac{\delta J}{\delta W^L} \\ \frac{\delta J}{\delta b^L} \end{bmatrix}$$

2.3.3 Activation functions

Activation functions that output small values for small value inputs and large outputs for large value inputs if that value reaches a threshold. This "non-linearity," where the output drastically changes at some threshold, introduces neural networks' ability to learn complex tasks.

2.3.3.1 ReLU

ReLU (rectified linear activation function) has the following property:

$$y_i = \begin{cases} x_i, & \text{if } x_i \geq 0 \\ 0, & \text{if } x_i < 0 \end{cases} \tag{2.1}$$

This property means that y should equal x unless it is lower than 0, and in that case, it is 0. ReLU is classified as a non-saturated activation function [51].

2.3.3.2 Leaky ReLU

Leaky ReLU is a variant of ReLU which will not set the negative values to 0. The definition is as following:

$$y_i = \begin{cases} x_i, & \text{if } x_i \geq 0 \\ \frac{x_i}{a_i}, & \text{if } x_i < 0 \end{cases} \tag{2.2}$$

where a_i is a constant in $\mathbb{Z}^+$ [51]. A problem with a regular ReLU is called *the dying ReLU*, which results in neurons becoming inactive, making all input result in an output of 0 [22]. Leaky ReLU solves this problem as it never changes any output to 0 [23]. Research suggests that leaky ReLU gives a better result than regular ReLU [51].

2.4 Normalization

The purpose of normalization is to bring different data sets to a similar scale. The reason is to equalize the impact of data points for the machine learning algorithm. For example, without normalization, data sets with huge numbers could overpower data sets with smaller numbers, even though both of their data could be equally important for the machine learning [32].

2.4.1 Min-max Normalization

Min-max normalization is a normalization technique where all values in a data-set are linearly transformed into fitting between a minimum and maximum value, such as 1 and 0. Thus, 0 Would correspond to the lowest value in the original data-set and 1 to the highest value. The function to calculate min-max normalization of a data point is as follows:

$$f(x) = \frac{x - x_{min}}{x_{max} - x_{min}}$$

where x_{min} is the original minimum value in the data set and x_{max} is the maximum value [32].

2.4.2 Z-Score Normalization

Z-Score normalization normalizes the data in a way that compensates for outliers. If a single value in a data-set is much higher or lower than the rest of the data points, using min-max normalization would skew the result, as almost all data points will be in the lower or higher section of the normalized data. Z-Score normalization solves this problem by using the mean and standard deviation. The created normalized data will hover around zero, depending on their standard deviation. The function for Z-Score normalization is as following:

$$f(x) = \frac{x - \mu}{\sigma}$$

where μ is the mean value of the data set, and σ is the standard deviation [32].

3
Previous work

Several researchers have published studies applying neural networks to forecast the stock market. Many have also done so together with financial indicators. This section presents relevant work to this thesis and how our approach differs from related work.

In the scope of input data for stock forecasting, most researches lean towards longer time segments, often using daily price increments [33, 18, 31]. Using daily prices provides a broader, more macroeconomic viewpoint for models to make predictions and requires an extended data collection period. Other research has used shorter time segments, in the order of minutes, [5, 40], in turn shortening data collection time. However, we have not found any research which attempts to utilize direct market orders, which show price jumps in real-time.

Numerous studies are published showing the use of machine learning for financial forecasting. Furthermore, the use of neural networks for this task has seen much research [5, 19]. For example, Yang et al. [3] and shows that the daily closing price of the Shanghai market can be predictable using artificial neural networks. Chong et al.[5] presents a deep neural network that utilizes historical returns from several stocks, showing with a sensitivity matrix that stock prices are to some degree correlated with each other. Hoseinzade et al. [16] also show that cross stock prices are correlated using a convolutional neural network (CNN), including time as an input dimension.

Patel et al. [34] create several machine learning networks for predicting stock market prices. They investigate the performance of combing different networks in a hybrid system. Using data from two different Indian indexes, they create an input vector containing ten financial indicators, such as RSI, MACD, and daily close. From their experimental results, the use of a two-stage fusion network reigns supreme. The first stage in the model consisting of a Support vector regression (SVR) following by a neural network. Patel et al. further explore using different machine-learning techniques but do not investigate how different indicators affect the result.

Agrawal et al. [1] further investigates the use of financial indicators for stock price prediction. Training an LSTM network using both volume and price indicators, they achieve high classification scores. Their experiments reveal that moving average and MACD correlate highly with the closing price. Dixon et al. [8] designs a deep neural network which input features consist of lagged price differences and moving averages

with varying window sizes. He concludes that DNN is a promising platform for stock price prediction combined with financial indicators.

As data size and data complexity increase, predictive models will too. Increasing local computational power has a limit, financial or physical. Distributed models serve as a solution to this problem, using the power from several computers in parallel. In 2014 Mu Li et al.[21] showed, using a parameter server, they could efficiently distribute a single model's parameters over several computation nodes. All nodes calculate gradients for different data samples in this system, where-after a server calculated new weights from all gradients, which are returned to the nodes.

In [14] Hajewski and Oliveira create an SmSVM distributed network for ensemble learning. Here several worker nodes create predictions that a single master node collects, and the final output is simply the majority vote of all worker predictions. Predictions are used as votes since all models need to approximate the same function; that is, the models try to predict the same target. Ahmed et al. [2] creates a similar network of machine learning models, creating a multi-model ensemble of machine learning models in order to forecast weather conditions.

4
Methods

This chapter explains the procedure that was applied to obtain our results. Furthermore, it describes utilized tools, libraries, and algorithms.

4.1 Data

All data used in this thesis is based on market orders from Nasdaq OMX Nordiq[30]. The data was gathered using their *Download CSV* options on the various stock overviews. Via this button, all daily market orders for each stock could be collected. Each market order contained price, time of trade, stock id, amount of shares etc. A script was written which collected this data daily, starting on the 25:th of August 2020 and continuing until the 19:th of April 2021. A CSV file for each stock combined the market orders, making it easy to process several months' worth of data.

4.1.1 Building features

The raw market orders obtained from the stock exchange need to be processed into features to be used with machine learning. This section describes what the input features for the neural networks contain and how the features are created.

4.1.1.1 Price

For each second during trading hours, all new market orders are placed into a fixed size queue. The size of this queue is referred to as window size for the prediction models. The values in this queue are appended to a CSV file, where each row corresponds to each second. This means that the CSV file contains the window size amount of market order prices for each second of open hours.

The window sizes tested are 70, 200, and 700. It should be noted that it is impossible to determine any time segment from the market orders, as it is unknown if the values in the queue would be replaced in one second or over several minutes.

4.1.1.2 Time

In the CSV file created, each row represents one second. A separate column for each price can be added to keep track of the time for each market order. The time is

normalized for each day. The formula for the normalization is the following:

$$F(x) = \frac{x}{32400}$$

where x is the timestamp adjusted to start at 0 and end at 32400. In other words the first timestamp of the day gets offset to become zero, the second timestamp one, and so on until the final timestamp of the day, which will become 32400.

4.1.1.3 Financial indicators

Several financial indicators were calculated in order to be used in neural networks. The financial indicators used were RSI, MACD, Volatility, EMA, and Price Channels. These indicators were calculated using the market orders described above. Many of these indicators describe change over a specific time interval, in which case several different time variations was calculated, for example, a 10-minute EMA and a 30-minute EMA.

Donchian channels inspired the implementation of the price channels, but the calculation of the max and min points differ. In order to find a max point for the price channels, two steps are performed. First, the time segment looked at is split into n number of sections. Secondly, the maximum points in the first and last section are selected, which is then used to calculate a straight line between the points in the following way:

$$y_1 = maxPoint_1$$
$$y_2 = maxPoint_2$$
$$x_1 = timestampMaxPoint_1$$
$$x_2 = timestampMaxPoint_2$$

$$k = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$
$$m = y - kx = y_1 - kx_1$$

After this, we can use the straight-line equation to calculate the y value of the line between the maximum points at the latest timestamp:

$$y = kx + m$$

where x is the current price at the latest timestamp.

4.1.2 Matching x and y data

As the input vector for the machine learning model cannot vary in size, there occurs a problem during the beginning of the day as the window might not be filled the first second upon market opening from a lack of market orders. The implemented solution is to wait until the window has been filled, then begin creating and saving features. In order to find the matching target value for each input vector, the latest

price is offset with the prediction time. When there is a new day, the window is cleared, and a new offset is created at that timestamp.

Another issue arises when trying to predict the future in the last n seconds of a day, where n is the number of seconds predicted into the future. As the last n seconds will not have any future prediction because the stock market has closed, the solution is to not predict during these seconds. However, by not predicting the last n seconds, it also means that the y data corresponding to the last n seconds of a day is not added to the data set.

4.1.3 Graphs

Plotting the stock price there are periodic straight lines in the y direction. The reason for this is when a weekend or end of the day occurs, and the price changes drastically during this time. This also means that the x-axis in the graphs only shows time when the stock market is open. Therefore the numbers on the x-axis should be seen as seconds of open stock-market time.

4.2 Distributed system

The distributed system is a system of nodes connected in any desired configuration. The nodes work together in layers, where each layer sends its output to the layer below it. For example, in the top layer, the nodes could get their data from a third-party source, such as a stock exchange. The nodes in the bottom layer output the final result of calculations in the system. The distributed system code was written in python to get a prototype up and running as fast as possible.

4.2.1 Node-Box

A Node-Box is the base piece of each node, as seen in Figure 4.1. This Node-Box contains a processor, inputs, and an output. For example, the processor could be an algorithm to calculate a financial indicator such as RSI or a machine learning algorithm. Each Node-Box belongs to a layer, where layer one is the top layer, and layer n is the bottom layer. The output of one or several Node-Boxes could be a single or several other Node-Boxes inputs in the layer below. Combining several Node-Boxes creates the possibility of setups that support redundancy, parallelisms, and increased computational speed. Figure 4.2 shows a simple example of a single Node-Box. In this figure, the inputs are the latest 200 market orders for the stock Swedbank.

Figure 4.3 shows a connected example of Node-Boxes. In this figure, there are a total of four nodes, three in layer one and one in layer two. All nodes in layer one

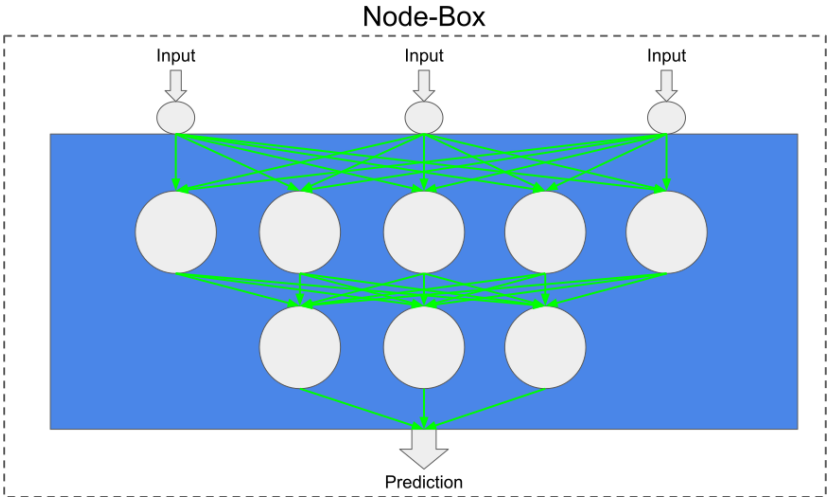


Figure 4.1: The base of a Node-Box

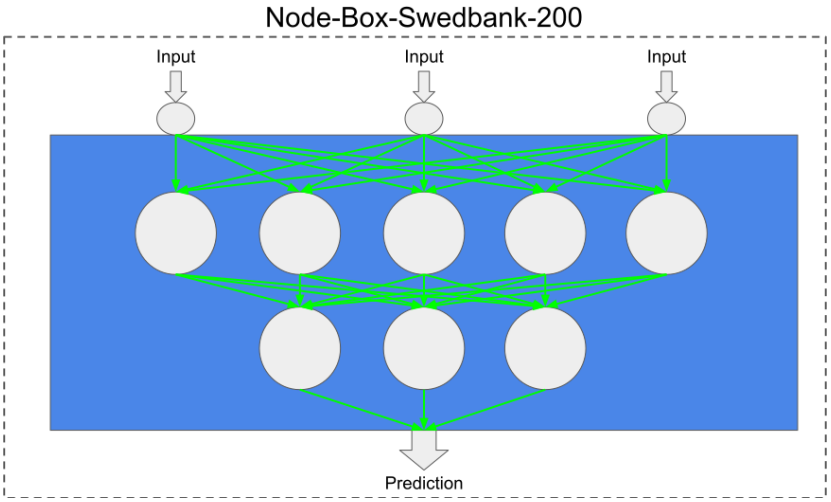


Figure 4.2: Node-Box with Swebank's 200 latest market orders

send their predictions as an input to *Node-Box-Swedbank-Final*, which calculates the final prediction with the help of all predictions from layer one.

4.2.2 Smart-sync

A data structure has been created to synchronize the different inputs received in a node-box, referred to as a*smart-sync* structure. The smart-sync synchronizes the data at different timestamps in an asynchronous way, making it fault-tolerant against delays.

As the predictions made by machine learning in the processing layers are between 5 and 60 seconds, it is more important that a prediction gets processed as fast as possible and less important that every prediction gets processed. For every second a prediction is not processed, it loses importance, and if more time has passed than

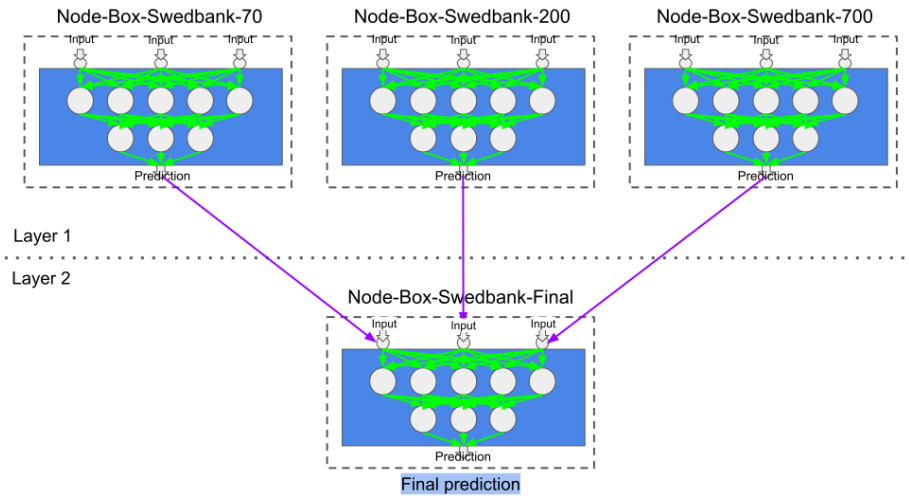


Figure 4.3: Three nodes in layer 1, with their targets connected to one node in layer 2

the prediction time, then we already know the actual price, and the prediction is useless. The smart-sync handles this issue by having a Window-Size (WS), which decides how many seconds the smart-sync should keep information.

A single matrix, with a height of the WS and a width of the *input-size+1*, creates the base for the smart-sync structure. The *input-size* is the size of the number of inputs that should be synchronized. Increasing the width by one makes it possible to fit a field to check how many cycles have been done. A floor division between the timestamp and WS calculates this field, denoted as *TS // WS*. Figure 4.4 visualizes the structure. In the figure, a field denoted *TS % WS* can be seen. This field is the index of the matrix, and *TS % WS* is how the data should be indexed. Thus, *TS % WS* is the timestamp modulo window-size and will place the new data in a cyclic pattern.

WS = 10				
TS % WS	TS // WS	Input1	Input2	Input3
0	0	NaN	NaN	NaN
1	0	NaN	NaN	NaN
2	0	NaN	NaN	NaN
3	0	NaN	NaN	NaN
4	0	NaN	NaN	NaN
5	0	NaN	NaN	NaN
6	0	NaN	NaN	NaN
7	0	NaN	NaN	NaN
8	0	NaN	NaN	NaN
9	0	NaN	NaN	NaN

Figure 4.4: Basic structure of the smart-sync.

When new data is to be added, three things are needed:

1. The timestamp
2. The node-box id
3. The data

The timestamp is used to calculate which row the data should be put in by using *TS % WS*. *The node-box id* is used to select which column the data should be placed in. The IDs need to be sequential, as to calculate the column position, the formula *(ID % input-size)+1* is used. Increasing the input size by one is used to ignore the first field where the number of cycles is kept track of. *The data* is added to the corresponding row and column, calculated from the timestamp and node-box id.

When data is added to the smart-sync two checks are done:

1. Compare the incoming data with the field of the current cycle. If the cycle is the same, add the data to the correct column. If the field is lower than the incoming data, erase all data in the row and add the new data to the corresponding column. If the field is higher than the incoming data, then do nothing. These steps will ensure that the new data overwrites the old data and that the old data never overwrites the new data.
2. Check if the row where the data was added is filled. If a row is filled, data has been received from all sources at the specific timestamp, which means that it can be processed. The smart-sync will thereby notify the processor with an array of the data at that timestamp.

Figure 4.5 shows an example of added data. The two rows in yellow have not been processed yet as they are still missing some data. To the left, under the column TS, shows the original timestamp. The upcoming timestamps would start overwriting the old ones, as shown in Figure 4.6. A darker green marks the overwritten rows from the old ones.

The second check when data is added to the smart-sync is tested in two different ways. In the first one, called Algorithm A, the whole row is iterated to check if it is filled. This iteration has a complexity of O(n). The second way, Algorithm B, is to have an integer that counts how many times an element has been inserted into a row. Every new cycle, the integer resets. Algorithm B has a time complexity of O(1), as there is no iteration.

	WS = 10				
TS:	TS % WS	TS // WS	Input1	Input2	Input3
1600000000 % 10 = 0	0	160000000	9.8	9.7	10.1
1600000001 % 10 = 1	1	160000000	10.1	9.9	10.4
1600000002 % 10 = 2	2	160000000	10	10.4	10.5
1600000003 % 10 = 3	3	160000000	9.8	10	10.1
1600000004 % 10 = 4	4	160000000	9.9	9.9	9.8
1600000005 % 10 = 5	5	160000000	10	10	11
1600000006 % 10 = 6	6	160000000	NaN	NaN	NaN
1600000007 % 10 = 7	7	160000000	9	9	9
1600000008 % 10 = 8	8	160000000	10.1	NaN	8.5
1600000009 % 10 = 9	9	160000000	10	10.5	10.6

Figure 4.5: Smart-sync with data added. The data in green is completed data that has been sent to be processed. The data in yellow is still missing data.

	WS = 10				
TS:	TS % WS	TS // WS	Input1	Input2	Input3
1600000010 % 10 = 0	0	160000001	10.2	10.3	10.9
1600000011 % 10 = 1	1	160000001	10.4	10.2	11
1600000012 % 10 = 2	2	160000001	10.3	10	10.9
1600000003 % 10 = 3	3	160000000	9.8	10	10.1
1600000004 % 10 = 4	4	160000000	9.9	9.9	9.8
1600000005 % 10 = 5	5	160000000	10	10	11
1600000006 % 10 = 6	6	160000000	NaN	NaN	NaN
1600000007 % 10 = 7	7	160000000	9	9	9
1600000008 % 10 = 8	8	160000000	10.1	NaN	8.5
1600000009 % 10 = 9	9	160000000	10	10.5	10.6

Figure 4.6: The dark green color indicates old data that has been overwritten.

4.2.3 Applying Techniques in Layer 1

Redundancy, parallelization, or an increase in processing power, can all be applied to layer one as well. For example, if an increase of processing power is desired, the number of Node-Boxes in layer one can be doubled and run in cycles the same way as layer 2. Figure 4.7 shows an illustration of this.

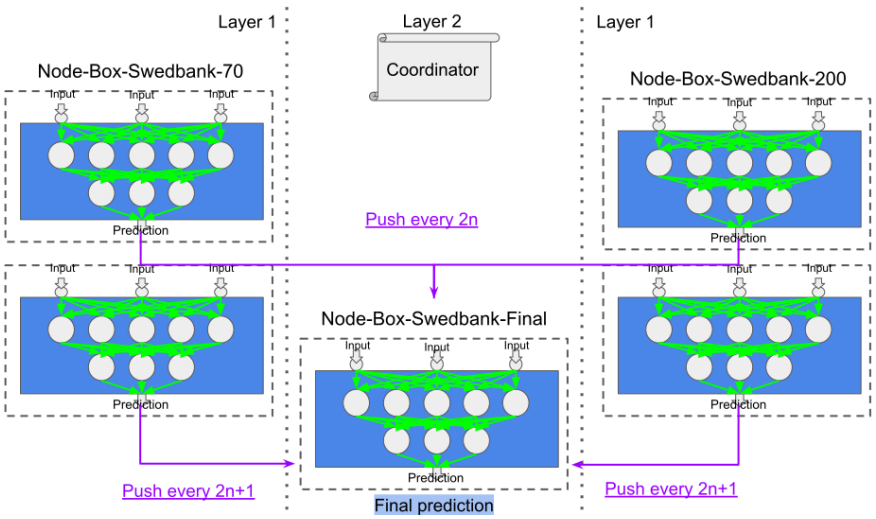


Figure 4.7: Nodes in layer 1 taking turns to calculate their target, possibly achieving higher processing power

4.2.4 Coordinator

The Coordinator is used to decide the communication between the different Nodes. The different nodes ask the Coordinator where they should send or receive their predictions, as seen in Figure 4.8. Depending on how the Coordinator is programmed, the nodes can be utilized in different ways. Some of the possible utilization could be for redundancy, parallelization, or an increase in processing power.

4.2.5 Coordinator Protocol

The protocol can be divided into three steps:

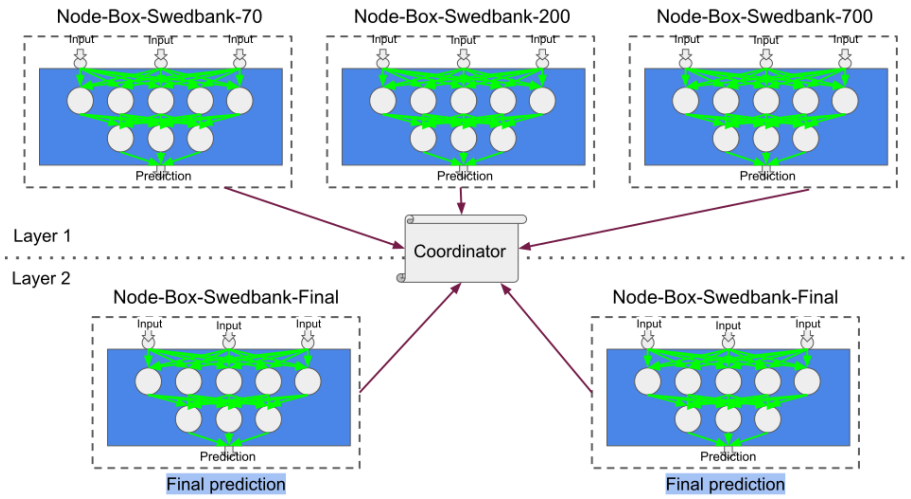


Figure 4.8: Node-Boxes asking the coordinator who and how to communicate

1. Discovery
2. Designation
3. Initiation

Discovery

In the *Discovery* phase, every node-box that wants to join the system connects to the coordinator and sends its layer position. The coordinator keeps track of the different node-boxes by saving the IP and port they used to connect. The coordinator can complete the *Discovery* phase in two different ways, depending on its configuration. The first way is to wait until a certain number of node-boxes has connected, and the second one is to have a waiting time and accept any number of node-boxes during that time.

Designation

In the *Designation* phase, the coordinator takes the different node-boxes discovered in the *Discovery* phase and assigns a connection between them. This connection is based on the configuration of the node-box, such as *Redundancy, Parallelization, or increased processing power*. When the coordinator has decided which node-boxes should communicate with which, it returns a key-value store with the following structure:

'port': int,
'id': int,
'server_ip_port': list(tuple)

port is the port number the node-box should use when hosting a server for its output. *id* is the unique id of the node-box, which can be used for identification. *server_ip_port* is a list of tuples, where each tuple contains an IP address and a port. The tuples are the node-boxes that the current node-box should listen to and use as input.

Initiation Lastly the *initiation* phase begins. In this step, the individual node-boxes has received their unique key-value store from the coordinator. As soon as this is received, the node-boxes will start their local server and try to connect to other servers in the *server_ip_port* key. As soon as all the node-boxes have connected and started their servers, the system is up and running.

4.2.6 Redundancy

Redundancy can be achieved by having two or more nodes in any layer with the same prediction model and the same input data. Any prediction of these nodes can then be used as the final answer. So, for example, if one of the nodes crashes, the other can continue to operate as usual. Figure 4.9 shows an example of this, where two Node-Boxes in layer two create redundancy.

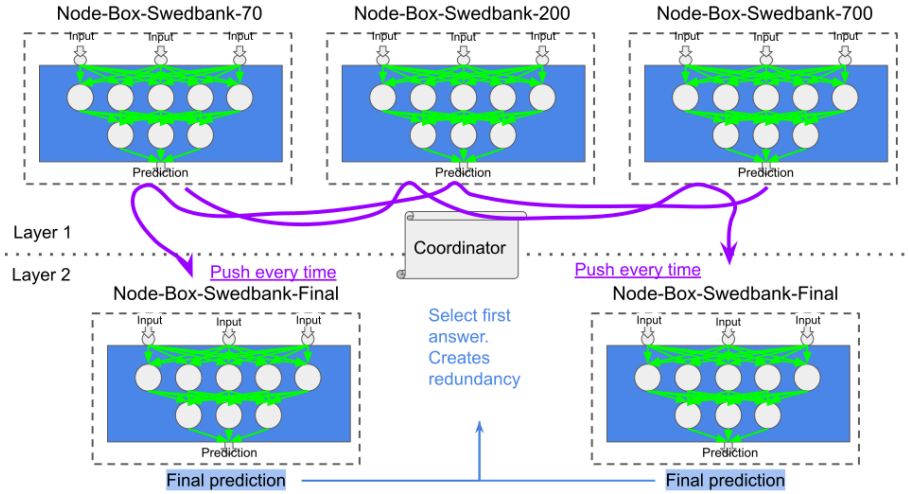


Figure 4.9: Two Node-Boxes are doing the same calculations to achieve redundancy.

4.2.7 Parallelization

It is possible to parallelize two different Node-Boxes by sending the same input to two or more Node-Boxes in any layer. The Node-Boxes will work independently on their data, and their predictions can be used as desired. Figure 4.10 shows an example of this with two Node-Boxes.

4.2.8 Cycling node-boxes

If the case would exist where a Node-Box cannot keep up with inference every second, it is possible to distribute this load over several Node-Boxes, which would increase the processing power. Every Node-Box would receive features to predict every n seconds, where the first Node-Box gets it on second n, the second one on

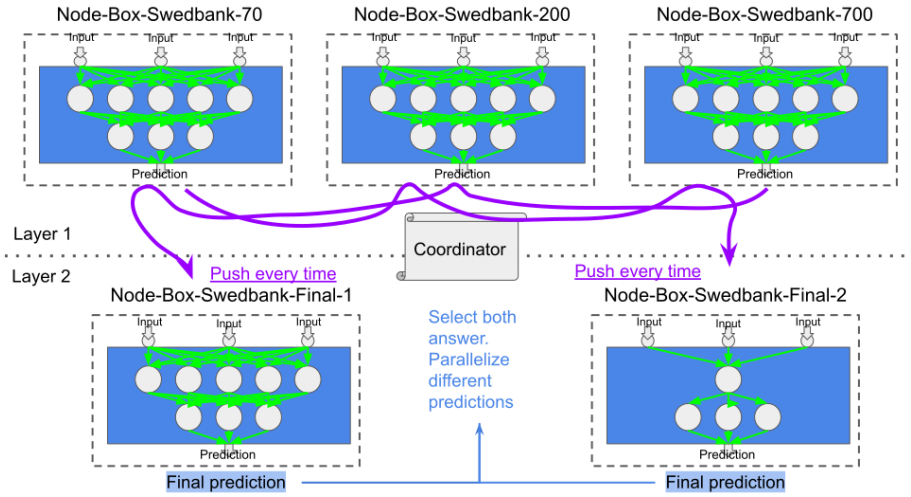


Figure 4.10: Two Node-Boxes doing different calculations to parallel processing.

second $n + 1$, and so on. When every node has received features to predict, it will restart again with node n. An example can be seen in Figure 4.11.

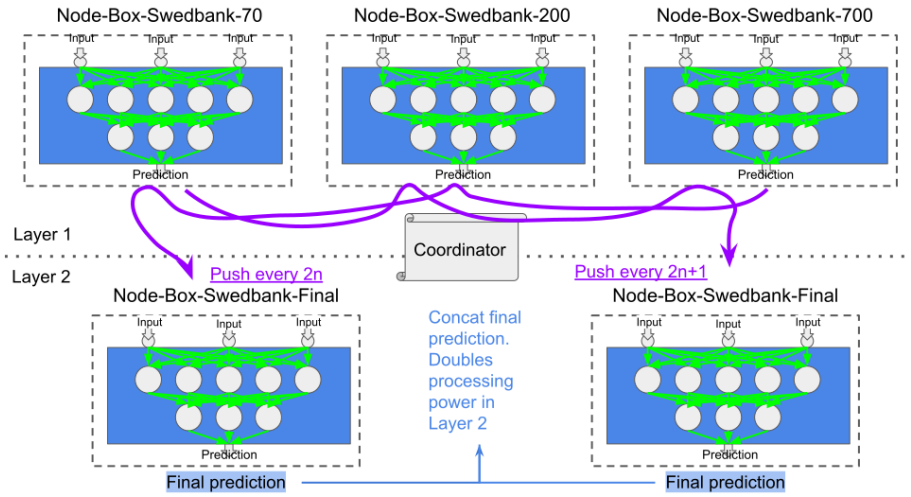


Figure 4.11: Two nodes cycling their workloads, increasing the processing power

4.2.9 Financial Indicator Node

The Financial Indicator Node (FI-Node) is a node that only calculates different Financial Indicators, such as MACD or RSI. The purpose is to add more features to any Node-Box in an effort to improve the machine learning performance. The FI-Node calculates financial indicators for any single stock, such as Swedbank, but could also calculate financial indicators for an index if desired. Figure 4.12 shows an illustration of this.

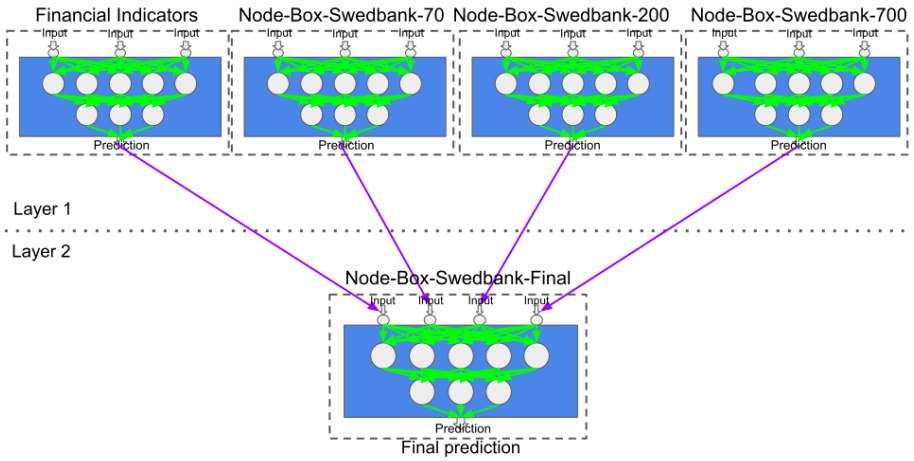


Figure 4.12: FI-Node, which gives financial indicators as features to the Node-Box in layer 2

4.2.10 Test Bench

When running any benchmarks or tests is was run on the system shown in Table 4.1.

CPU	Intel i5-4670k (4 cores), Overclocked to 4,3 GHZ
GPU	Nvidia GTX 970
RAM	32 GB DDR3
OS	Pop!_OS 20.10 (based upon Ubuntu 20.10)

Table 4.1: The hardware specification of the test bench.

4.3 Stock predictor

This section describes the machine learning libraries, input feature construction, and network architecture used in this thesis. It further describes how the training for layer one models and created and describes the combined distributed system.

4.3.1 PyTorch

The machine learning library, PyTorch, creates and trains models. PyTorch is an open-source library that Facebook mainly develops. This thesis uses the python interface but is executed in a C++ environment. PyTorch introduces a tensor data structure, a matrix-style structure developed for fast computations on graphic processing units (GPU). It also contains an auto differentiation feature, which alleviates training neural networks through backpropagation[35].

4.3.2 Neural networks

There are three neural networks used in this thesis denoted *The deep network*, *The shallow network*, and *The combiner network*. *The deep network* consists of 5 linear layers, where the activation function of the layers is Leaky ReLU. The layers get gradually smaller as they approach the final layer. Listing 4.1 shows the implementation of *The deep network*. *The shallow network* consists of 3 linear layers, where the first layer increases the number of neurons, but the other two gradually decrease them. The activation function for each layer is Leaky ReLU. Listing 4.2 shows the implementation of *The shallow network*. Leaky ReLU is used instead of a regular ReLU to avoid the problem of *dying ReLU*. Both *The deep network* and *The shallow network* are only used for single stock prediction. For a combined stock prediction, *The combiner network* is used. This network' has similar depth to *The shallow network*, but its width is dependant on the input size. Listing 4.3 shows the network in detail.

4.3.3 Input data

The input vector containing all features for the neural networks is a combination of the market order price, market order publication time, and financial indicator data. Naturally, different models require different combinations of these data features, and might therefore include or exclude some of the features, however, the general shape of feature vector X is the following:

$$X = \Big[p_0, ..., p_w, t_0, ..., t_w, EMA_a, RSI_b, MACD_c, Vol_d, PC_e^{minK}, PC_e^{maxK}, PC_e^{minY}, PC_e^{maxY}\Big] \quad (4.1)$$

where w is the window size, p_i is the price from the market order that was added $w-i$ market orders ago. The market order publication time is represented as t_i, meaning that p_i and t_i stem from the same market order. $EMA_a, RSI_b, MACD_c, Vol_d$ are financial indicators where a, b, c and d represent different time windows for which the indicators are applied. The price channel is described with PC_e^{minK}, PC_e^{maxK}, PC_e^{minY}, PC_e^{maxY}, where $minK$ and $maxK$ is the slope for the top and bottom channel, and where $minY$ and $maxY$ are y-axis values for points on the bottom and top channel lines.

The non-distributed models *The deep network*, *The shallow network*, also referred to as layer one models, requires input vectors of the shape described by Equation 4.1. The *The combiner network* uses outputs from layer one models together with financial indicator data to create is input vector X_d with the shape:

$$X_D = \Big[p_{70}, p_{200}, p_{700}, EMA_a, RSI_b, MACD_c, Vol_d, PC_e^{minK}, PC_e^{maxK}, PC_e^{minY}, PC_e^{maxY}, p_N\Big] \quad (4.2)$$

where p_{70}, p_{200}, p_{700} is the outputs of layer one models using 70, 200 and 700 market order window sizes and p_N is cross stock data from a Nordea predictor.

4.3.4 Output data

The output from the models is stock price, meaning that the models will be aiming to solve a regression task. Time to predict into the future, or prediction time, determines how far ahead the model should predict. For this thesis we will aim for 30 second prediction times, motivated by the real-time nature of using market orders as input data.

Listing 4.1: *The deep network*

```
class DeepModel(nn.Module):
    def __init__(self):
        super().__init__()

    def instantiate(self, input_size):
        self.fc1 = nn.Linear(input_size, input_size*2).type(dtype)
        self.fc1.weight.data.uniform_(-0.1, 0.1)
        self.fc2 = nn.Linear(input_size*2, round(input_size*1.5)).type(dtype)
        self.fc2.weight.data.uniform_(-0.1, 0.1)
        self.fc3 = nn.Linear(round(input_size*1.5), round(input_size*0.5)).type(dtype)
        self.fc3.weight.data.uniform_(-0.1, 0.1)
        self.fc4 = nn.Linear(round(input_size*0.5), 20).type(dtype)
        self.fc4.weight.data.uniform_(-0.1, 0.1)
        self.fc5 = nn.Linear(20, 1).type(dtype)
        self.fc5.weight.data.uniform_(-0.1, 0.1)

    def forward(self, x):
        x = F.leaky_relu(self.fc1(x))
        x = F.leaky_relu(self.fc2(x))
        x = F.leaky_relu(self.fc3(x))
        x = F.leaky_relu(self.fc4(x))
        x = F.leaky_relu(self.fc5(x))
        return x
```

Listing 4.2: *The shallow network*

```
class ShallowModel(nn.Module):
    def __init__(self):
        super().__init__()

    def instantiate(self, input_size):
        self.fc1 = nn.Linear(input_size, input_size*3).type(dtype)
        self.fc1.weight.data.uniform_(-0.1, 0.1)
        self.fc2 = nn.Linear(input_size*3, round(input_size*0.5)).type(dtype)
        self.fc2.weight.data.uniform_(-0.1, 0.1)
        self.fc3 = nn.Linear(round(input_size*0.5, 1)).type(dtype)
        self.fc3.weight.data.uniform_(-0.1, 0.1)

    def forward(self, x):
        x = F.leaky_relu(self.fc1(x))
        x = F.leaky_relu(self.fc2(x))
        x = F.leaky_relu(self.fc3(x))
        return x
```

Listing 4.3: *The combiner network*

```
class CombinerModel(nn.Module):
    def __init__(self, input_size):
        data_type = torch.cuda.FloatTensor
        super().__init__()
        self.fc1 = nn.Linear(input_size, input_size*3).type(data_type)
        self.fc1.weight.data.uniform_(-0.1, 0.1)
        self.fc2 = nn.Linear(input_size*3, round(input_size)).type(data_type)
        self.fc2.weight.data.uniform_(-0.1, 0.1)
        self.fc3 = nn.Linear(round(input_size), 1).type(data_type)
        self.fc3.weight.data.uniform_(-0.1, 0.1)

    def forward(self, x):
        x = f.leaky_relu(self.fc1(x))
        x = f.leaky_relu(self.fc2(x))
        y = f.leaky_relu(self.fc3(x))
        return y
```

4.3.5 Training

When training the non-distributed models, the data set becomes very large, surpassing the system memory available in the test system. Since all the data could not be read into memory at once, chunking was applied. When training with chunks, the data-set is split into several chunks of a chosen size. The size used for the chunks was 500 000 rows, which made it possible to fit any data-set used into the RAM size of 16GB. When training a model, each chunk is trained iteratively. When the first chunk is read, the model is trained for the entirety of the epoch range. After that, the next chunk is read. This process is repeated until all chunks have been processed. The *Combiner model* did not face this problem as its training data contained substantially smaller input feature vectors, directly decreasing the data memory size. Therefore, layer two models using the *Combiner model* did not employ chunking when training.

The distributed system does not contain an implementation to train all machine learning nodes based on the final predictions of the last layer. Each machine learning model needs to be trained separately, where all layers, except the top layer, trains with the output from previous layers' test data.

4.3.6 Testing

The comparison of different networks is measured by the loss and by comparing a graph of the predicted data with the original data. Because of how loss is measured, normalized and non-normalized data will not be directly compared using loss. If any comparison would be made between different stocks, the loss is likely to be partially misleading depending on the difference in price, volatility, or other indicators. Therefore all conclusions made from loss only apply to the specific stock and the normalized/non-normalized versions. Only optical measurements between the graphs will be done to get an idea of the performance between stocks or between normalized and non-normalized data. Additional metrics were developed for comparison reasons. A 10-minute average prediction strategy was employed as a benchmark. This prediction strategy predicts using the 10-minute average price. Another strategy that was used was *O set*, where the prediction is the current price, thus producing a perfect 30-second offset if the target prediction time is 30-second.

Exhaustive testing will be done to measure the performance of different parameters. A script testing all the different parameters will run and save the result to a spreadsheet, making it easy to compare the different configurations.

- 151 days for training in layer 1 (Aug 25 2020 - 22 Jan 2021)
- 37 days for evaluation in layer 1 (23 Jan 2021 - 28 Feb 2021)
- 49 days for testing in layer 1 (1 Mars 2021 - 19 April 2021)
- 37 days for training in layer 2 (1 Mars 2021 - 7 April 2021)
- 6 days for evaluation in layer 2 (8 April 2021 - 13 April 2021)
- 6 days for testing in layer 2 (14 April 2021 - 19 April 2021)

The 49 days for testing in layer 1 is the same days that is used in training, evaluation and testing in layer 2. The reason for this is to utilize as much data as possible. Figure 4.13 visualizes the split.

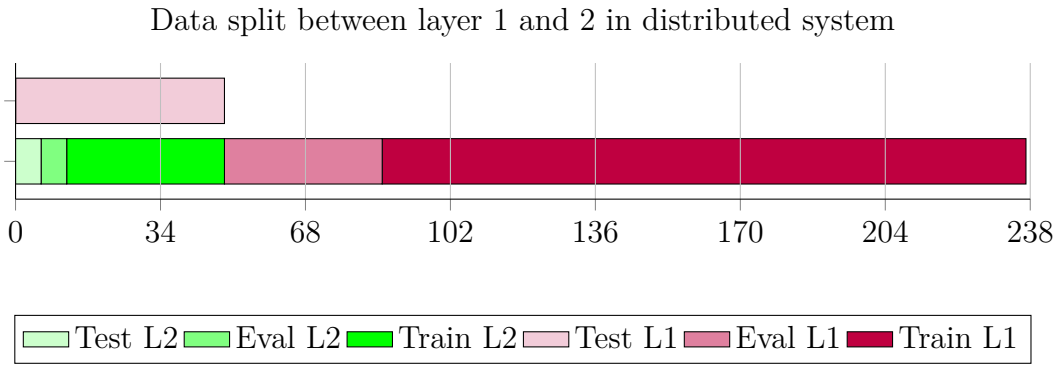


Figure 4.13: Graph showing how the data is split in the distributed system. Test L1 overlaps Train L2, Eval L2, and Test L2 as the same dataset is used. Lx means Layer x.

5
Results

5.1 Feed-forward Neural Network

This section presents the results and discussion of the single stock predictions. The models are trained in different ways, where variables such as input data and learning rates differ. Therefore a naming convention is used. The model name *70_100e_lr0.0001_S* means that the model used 70 market orders as input data, was trained for 100 epochs, used a learning rate of 0.0001, and *S* stands for Swedbank. If the model is trained and used for Nordea stocks, the last symbol is *N* instead.

In table 5.2 and table 5.3 loss scores for different prediction strategies are presented.The strategy called *10AVG* is the 10-minute average prediction strategy and *O set* the 30-second offset strategy.

5.1.1 Min-max normalization

Using min-max normalization over the entire data-set of Swedbank data had mixed results. The results presented are from the best models for each window size investigated: 70, 200, and 700. The presented losses are from test data between the period 1 Mars to 19 April.

(a) 70_100e_lr0.0001_S **(b)** 200_50e_lr0.0001_S **(c)** 700_100e_lr0.0001_S

Figure 5.1: Graphs for predictions of three models using min-max normalized data with different window sizes; 70, 200 and 700. All graphs depict the same time period

Figure 5.1 depicts three models predictions over test data: *70_100e_0.0001_S*, *200_50e_lr0.0001_S* and *700_100e_lr0.0001_S*. The first model,

Model	MSE_preds	MAE_preds	MSE_offset	MAE_offset
70_100e_lr0.0001_S	0.06316	0.21146	0.00310	0.03156
200_50e_lr0.0001_S	0.01813	0.10073	0.00313	0.03147
700_100e_lr0.0001_S	0.94534	0.88968	0.00328	0.03011

Table 5.1: MSE and MAE test-set losses for models using min-max normalization. Swedbank between 1 Mars - 19 April

70_100e_0.0001_S, uses a window size of 70 market orders together with time data for the market orders. This is the only model of the three that uses time, as the result of the other two models with time was inadequate. Its prediction follows the general price trend but fails to generalize accurately. Graph **b** shows a model with window size 200 and no time data, which follows the price trend better. Moreover model *200_50e_lr0.0001_S* has the lowest MSE and MAE loss, as seen in Table 5.1, of all three models shown in Figure 5.1. The larger model *700_100e_lr0.0001_S* performs poorly, failing to follow the general price trend and shows the highest loss, both in MSE and MAE loss. None of the models using min-max normalization outperformed the offset score, shown in Table 5.1.

5.1.2 Z-normalization

The Z-normalized data consists of market data from Swedbank and Nordea stocks. The machine learning for Swedbank and Nordea is trained with a window size of 70, 200, and 700 market orders, respectively.

5.1.2.1 Swedbank

The model *70_100e_lr0.0001_S*, which uses a 70 market order window, follows the price trend well, seen in the left graph of Figure 5.2. It is, however, offset in the x-axis, seen in the 30-minute graph in Figure 5.2. Figure 5.3 shows the prediction of Swedbank with a window size of 200 market orders. With a window size of 200 market orders, Swedbank still follows the price trend, but less tightly compared to a window size of 70. It is not necessary to look at the zoomed-in graph in Figure 5.3 (a) to see the offset in the x-axis. However, Figure 5.3 (b) shows that the predictions are not very accurate and that it misses some dips that the 70 market order model was able to predict. Figure 5.4, depicting the prediction graph for model *700_100e_lr0.0001_S*, shows that the trend of following the target graph less tightly continues. The zoomed-in version seen in Figure 5.4 (b) clearly shows that the prediction does not follow small changes in the target data.

(a) 3 hour window **(b)** 30 min window

Figure 5.2: Two graphs for Swedbank stock price prediction using model 70_100e_lr0.0001_S. Left graph shows prediction for 3 hour window of Swedbank stock. The left shows a 30min window. Larger figures can be found in Appendix I.

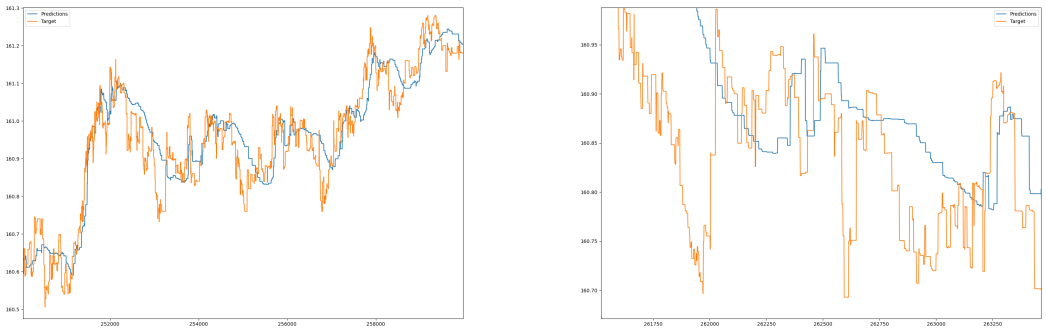
Parameters	MSE_preds	MAE_preds	MSE_10AVG	MAE_10AVG	MSE_Offset	MAE_Offset
70_100e_lr0.0001_S	0.00277	0.03598	0.01518	0.09137	0.00326	0.03359
200_50e_lr0.0001_S	0.00877	0.06854	0.01497	0.09090	0.00327	0.03348
700_100e_lr0.0001_S	0.01314	0.08675	0.01312	0.08671	0.00343	0.03220

Table 5.2: Table showing loss scores over Swedbank test set, 1 Mars to 19 April for three different models. Losses for a 10 minute average strategy and the offset strategy is also shown. Swedbank single models (the best ones, used in dist) 70 uses time, 200, 700 does not (Deep 30s all)

Table 5.2 shows losses of the different models predicting Swedbank. The model *70_100e_lr0.0001_S* performs best, beating all other models, including the 10-minute average and offset strategy. On the other hand, models *200_50e_lr0.0001_S* and *700_100e_lr0.0001_S* does not perform on par with the 70 market order model. Moreover, the models show higher loss than the offset strategy, and in the case of *700_100e_lr0.0001_S* higher MAE loss than the 10-minute average.

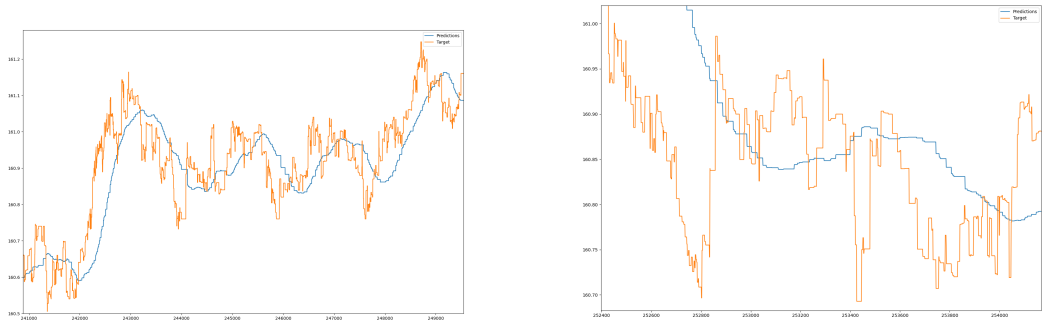
5.1.2.2 Nordea

Figure 5.5 depicts prediction windows for 3 hours and for 30 minutes using model *70_100e_lr0.0001_N*. In the 3-hour window graph, the model follows the price trend well, missing some local price jumps. In the 30-minute window, an x-axis offset becomes evident, but the model still follows price movements. Despite this offset, the model performs well, outperforming other Nordea models in both MSE and MAE loss, seen in Table 5.3. The 200 market order window model *200_100e_lr0.0001_N* follows the price movement in both time windows shown in Figure 5.6. However, prediction movements are more conservative compared to model *70_100e_lr0.0001_N*. The 30-minute window in Figure 5.6 (b) shows smaller prediction movements, where large price jumps are more constrained. Model *700_30e_lr1e-05_N* uses 700 market orders and its performance is seen in Figure 5.7. The 3-hour graph shows a rough price movement following but with a larger x-axis offset compared to



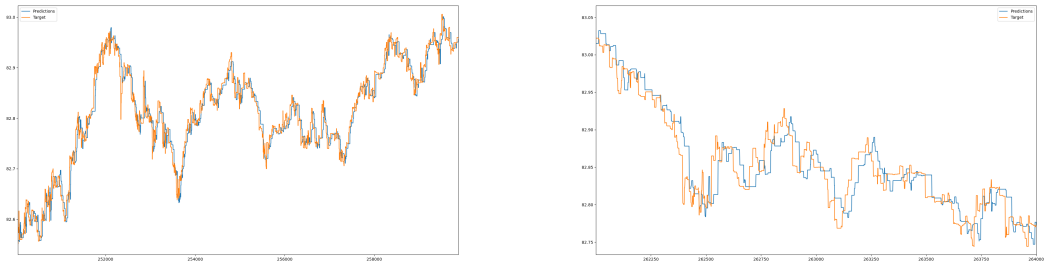
(a) 3 hour window **(b)** 30 min window

Figure 5.3: Two graphs for price prediction using model 200_50e_lr0.0001_S. Left graph shows prediction for 3 hour window of Swedbank stock. The left shows a 30min window. Larger figures can be found in Appendix I.



(a) 3 hour window **(b)** 30 min window

Figure 5.4: Two graphs for price prediction using model 700_100e_lr0.0001_S. Left graph shows prediction for 3 hour window of Swedbank stock. The left shows a 30min window. Larger figures can be found in Appendix I.



(a) 3 hour window

(b) 30 min window

Figure 5.5: Two graphs for price prediction using model *70_100e_lr0.0001_N*. Left graph shows prediction for 3 hour window of Nordea stock. The left shows a 30min window. Larger figures can be found in Appendix I.

70_100e_lr0.0001_N and *200_100e_lr0.0001_N*. The 30-minute window shows a failure to react rapidly to local price movements.

Parameters	MSE_preds	MAE_preds	MSE_10AVG	MAE_10AVG	MSE_Offset	MAE_Offset
70_100e_lr0.0001_N	0.00079	0.01786	0.00432	0.04758	0.00072	0.01652
200_100e_lr0.0001_N	0.00124	0.02188	0.00426	0.04734	0.00073	0.01643
700_30e_lr1e-05_N	0.00204	0.03310	0.00385	0.04543	0.00086	0.01588

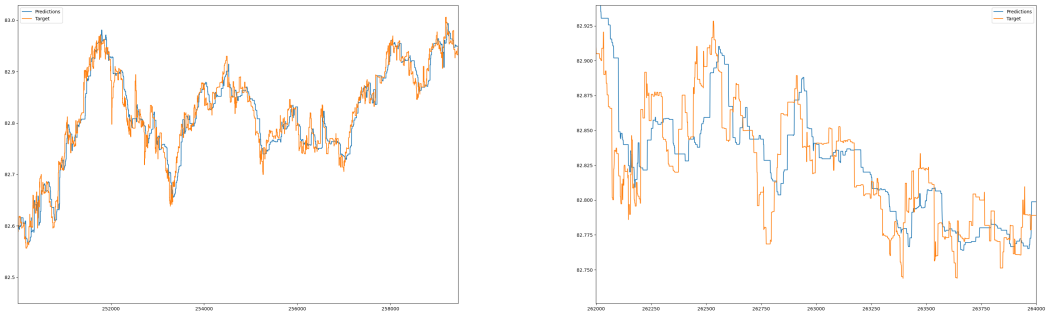
Table 5.3: Table showing loss scores over Nordea test set, 1 Mars to 19 April for three different models. Losses for a 10 minute average strategy and the offset strategy is also shown.

Table 5.3 shows the loss results for the Nordea stock test set. Model *70_100e_lr0.0001_N* outperforms all other models in the table, including both MSE and MAE with 10-minute running average, but only beats the offset strategy in terms of MSE loss. On the other hand, the offset strategy outperforms *70_100e_lr0.0001_N* in MAE. The models *200_100e_lr0.0001_N* and *700_30e_lr1e-05_N* both received lower losses than the 10-minute running average but higher than the offset losses.

5.1.3 Discussion

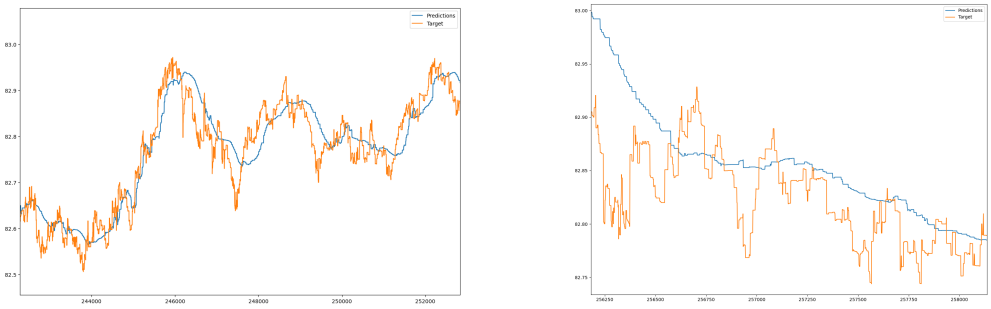
As shown in the results above, using more than 70 market orders as input data does not seem to increase prediction accuracy. The models using 70 market orders also contain the time data for each market order, so at first glance, this seems to be the separating factor for success. However, accuracy faltered when using time data combined with models using 200 or 700 market order inputs.

Models using 70 market orders as input performed well against the 10-minute average and offset prediction strategy but did not consistently outperform them. The close results to the offset losses are especially interesting since the model's prediction graphs are skewed in the x-axis, becoming similar to the offset predictions. This



(a) 3 hour window **(b)** 30 min window

Figure 5.6: Two graphs for price prediction using model *200_100e_lr0.0001_N*. Left graph shows prediction for 3 hour window of Nordea stock. The left shows a 30min window. Larger figures can be found in Appendix I.



(a) 3 hour window **(b)** 30 min window

Figure 5.7: Two graphs for price prediction using model *700_30e_lr1e-05_N*. Left graph shows prediction for 3 hour window of Nordea stock. The left shows a 30min window. Larger figures can be found in Appendix I.

result indicates that the models have a hard time generalizing patterns in the price data and perhaps sometimes predict safely, that is, its latest market orders price. However, as seen for the Swedbank single stock model, it is possible to outperform the offset strategy, as Swedbank single stock model indicates learned patterns.

Normalizing time series data contains some complications. Traditionally one normalizes the input over the entirety of a data set, for instance, using min-max normalization. Doing such a normalization, however, can introduce problems. Firstly, future data higher or lower than the normalized data set will not fit inside the normalization range. Furthermore, if normalizing the entire data set with parameters calculated from the entire data set, some information about future data is leaked. However, this possible leak does not seem to impact our results as normalizing in this way performs worse than normalizing according to a sliding time window. A reason for this could be that when normalizing over the entire data set, a larger span of values need to be accounted for, making the normalized values possibly very small. Since the networks already showed instability issues, using very small values could further cause this effect, in turn decreasing performance.

5.2 Distributed System

The first two subsections cover the results from the smart-sync and node-boxes. In the two subsections after, the results from the smart-sync and node-boxes are discussed.

5.2.1 Smart-sync

The smart-sync is tested by creating six different instances with input sizes of 10, 100, 1000, 10 000, 100 000, and 1 000 000 elements. A mean is calculated by running the test ten times for each instance. The test works by linearly putting data into one row until filled. The test compares two different algorithms. Algorithm A is an algorithm that iterates through every row to see if all the positions are filled. Algorithm B has an integer for every row that keeps track of the number of filled elements by incrementing the integer every time anything is put into the row. As the test adds n elements, Algorithm A gets a complexity of $O(n^2)$ during the test, and Algorithm B a complexity of $O(n)$. The results can be seen in Figure 5.8 and Table 5.4.

5.2.2 Node-Boxes latency

In order to measure the latency of the node-boxes a specific scenario is needed. In the chosen scenario, many node-boxes are in layer one, and one node-box is in layer two. All node-boxes in layer one connect to the one in layer two. The latency is measured from when each node in layer one starts to process a market order at a given timestamp and stops when the node-box in layer two has predicted this timestamp. Running all the node-boxes on the same computer creates a best-case

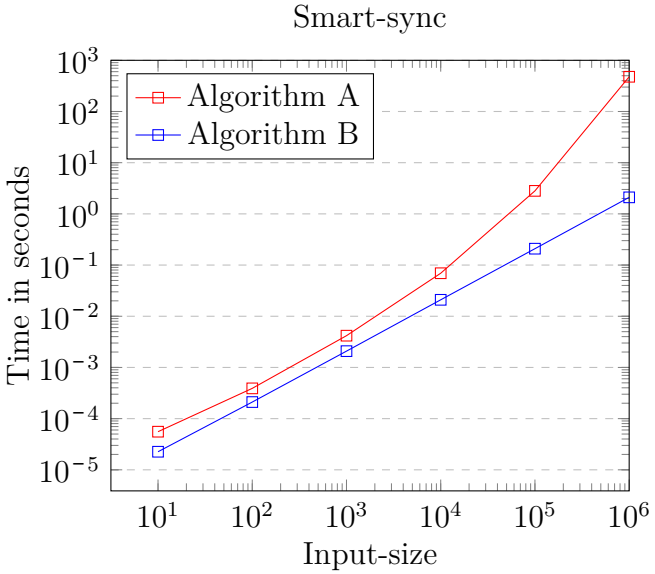


Figure 5.8: Graph showing the difference between Algorithm A and B.

Input-size	Algorithm A (seconds)	Algorithm B (seconds)
10	0.0000555754	0.0000225544
100	0.0003901720	0.0002108097
1000	0.0041720390	0.0020816088
10 000	0.0692465067	0.0209511042
100 000	2.8141211271	0.2088546753
1 000 000	477.0214994431	2.1033621311

Table 5.4: Comparison between Algorithm A and B with different input-sizes.

scenario, where each node-box runs in an individual process. This way, there will not be any latency from any network connection between computers. To get the lowest latency, smart-sync used *Algorithm B*.

Five configurations specify the test, which are 1, 2, 4, 6, and 8 node-boxes in layer one. The mean, fastest, and slowest latency is measured during 5 minutes, and Figure 5.9 displays the result. The first 10 seconds of the tests are not included, as they were unstable and not representative over a more extended period.

	1:1 Boxes	2:1 Boxes	4:1 Boxes	6:1 Boxes	8:1 Boxes
Slowest time (s)	0.005547	0.006376	0.007149	0.007043	0.007882
Fastest time (s)	0.001353	0.001203	0.001490	0.001223	0.001306
Mean time (s)	0.00332	0.003262	0.003531	0.004311	0.004945

Table 5.5: Node-boxes benchmark as $n : m$, where n is the number of nodes in layer 1, and m the number of nodes in layer 2

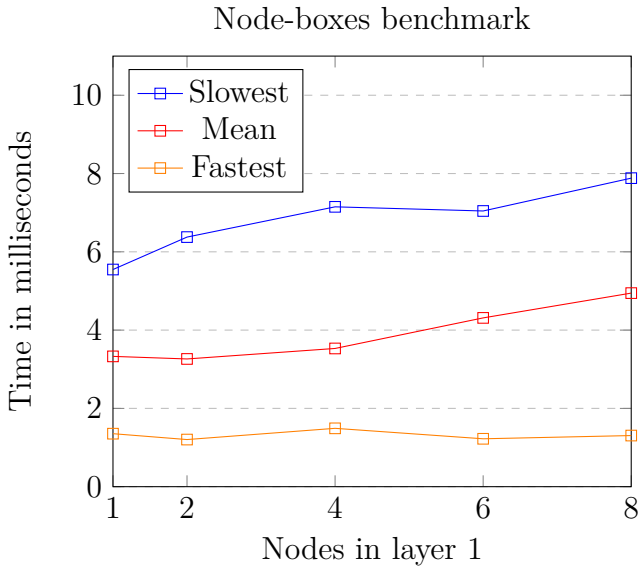


Figure 5.9: Graph showing the fastest, mean, and slowest time it took to run n node-boxes in layer 1, with one node-box in layer two. The test is done over a 5 minute interval and is measured from the time a node in layer 1 starts its processing, until the node in layer 2 calculates it prediction from that timestamp.

5.2.3 Discussion of Smart-sync

When looking at the results from the smart-sync, the superior algorithm is *Algorithm B*. It is more than two times faster when working with an input size of 10 and more than 200 times faster when working with 1 000 000 inputs. It should, however, be considered that during the testing done throughout this thesis, when using node-boxes, the input size has been in the 10^1 area. With such a small input size, we would compare a speed of 55.5754 µs in Algorithm A against the speed of 22.5544 µs in Algorithm B. With such fast speeds, the result could be seen as negligible, as when compared to the results of the node-boxes, the fastest one of all results was 1203 µs. Adding the latency delta of Algorithm A and B to the result from the node-boxes would result in a speed of 1236.0210 µs instead.

With an input size more prominent than 10^5, there is a dramatic performance increase by using Algorithm B. The likelihood of applying such an extensive network might be small, but there could potentially be applications that could use such a smart-sync structure.

5.2.4 Discussion of Node-boxes

The results from the node-boxes show that the meantime is in the millisecond range, as seen in Table 5.5. The table shows that the meantime for 1, 2, and 4 nodes is very similar, but it increases more rapidly for 6 and 8 nodes. This increase is possibly from the CPU having four cores and splitting the work more evenly for up to 4 node-boxes. The explanation for two node-boxes being faster than one is probably

a random margin of error.

To get an idea of how fast the processing is, the slowest time, 0.007882 seconds, can be taken in comparison to a future prediction of 30 seconds, where the delay in percent can be calculated:

$$\frac{0.007882}{30} = 0.000262733 \approx 0.026\%$$

This result shows that the time delta from the prediction start until finished has a 0.026% delay of 30 seconds to make the prediction. As the prediction's result gets less relevant from each second that passes (after 30 seconds, the prediction is useless, as the actual price is then known), having this slight delay of only a few milliseconds is a success. This tiny delay would even make it possible to predict only a few seconds into the future.

With such low latency, the question arises of how valuable cycling node-boxes are, as written about in Section 4.2.8. If one node-box can process the data from any timestamp in less than one second, it will always keep up, but otherwise, it will always become more and more delayed. However, three things have been identified that could increase the latency:

- Worse hardware. Currently, a GTX 970 is used, which is a dedicated graphics card. Using a weaker graphics card or a CPU for the machine learning inference would increase processing times.
- More complex processing algorithm. The current neural network could become more complex, which would increase the processing times.
- Processing more often than every second. Right now, the latest trade during a timestamp for each second is used. Splitting this second into even smaller timestamps would require more processing per second, and therefore shorter time available for processing.

The processing times might surpass the processing time window by meeting any or all of these three constraints, and therefore making the *cycling node-boxes* useful.

5.3 Distributed combined models

This section presents the prediction results from the distributed network. The models with a name ending with **D** are distributed models. The method chapter presents the input features into the second layer models.

Model	MSE_preds	MAE_preds	MSE_10AVG	MAE_10AVG	MSE_Offset	MAE_Offset
vol50_lr0.01_None_D	0,00163	0,02803	0,01023	0,07745	0,00294	0,02832
rsi30_lr0.01_None_D	0,00164	0,02830	0,01023	0,07745	0,00294	0,02832
ema15_macd_rsi5_rsi30_vol100_vol50_lr0.001_Nordea70_D	0,00164	0,02833	0,01023	0,07745	0,00294	0,02832
rsi5_rsi30_vol100_vol50_ema30_ema15_lr0.001_Nordea70_D	0,00164	0,02834	0,01023	0,07745	0,00294	0,02832
macd_lr0.01_None_D	0,00165	0,02828	0,01023	0,07745	0,00294	0,02832
ema30_ema15_macd_rsi5_rsi30_vol100_lr0.001_Nordea700_D	0,00165	0,02831	0,01023	0,07745	0,00294	0,02832
vol100_lr0.01_None_D	0,00165	0,02832	0,01023	0,07745	0,00294	0,02832
ema15_lr0.001_NordeaPred200_D	0,00165	0,02837	0,01023	0,07745	0,00294	0,02832
rsi30_vol100_vol50_ema30_ema15_macd_lr0.001_Nordea200_D	0,00165	0,02837	0,01023	0,07745	0,00294	0,02832
rsi5_lr0.01_None_D	0,00165	0,02844	0,01023	0,07745	0,00294	0,02832

Table 5.6: Table shows MSE and MAE losses for layer two models used in the distributed network. Sorted by MSE loss. Data is Swedbank stock price for 13 April - 19 April

Table 5.6 shows the results for the ten best performing distributed models. The table shows that no single model outperforms the rest. However, model *vol50_lr0.01_None_D* does have slightly lower MSE and MAE losses compared to the others. Furthermore, model *vol50_lr0.01_None_D* and model *rsi30_lr0.01_None_D* are the only models that beat the offset strategy in MAE loss. All models beat MSE and MAE for 10-minute average strategy and MSE for offset strategy.

Model	MSE	MAE
vol50_lr0.01_None_D	0,00163	0,02803
rsi30_lr0.01_None_D	0,00164	0,02830
70_100e_lr0.0001_S	0.00170	0,02889
200_50e_lr0.0001_S	0,00648	0,05812
700_100e_lr0.0001_S	0,01889	0,07789

Table 5.7: Table shows MSE and MAE losses for top performing distributed models and single stock models. Data is Swedbank stock price for 13 April - 19 April

Table 5.7 compares the top distributed models with the single stock models used in the input feature vector. Losses in this table are calculated from the period 13 April to 19 April. All single stock predictors perform slightly worse than the distributed models, measured in MSE and MAE losses.

Figure 5.10: Graph shows stock price predictions for the distributed model *vol50_lr0.01_None_D*. Pred70, Preds200 and Preds700 refers to the prediction inputs used for the distributed model. Predictions for a 3 hour Swedbank window.

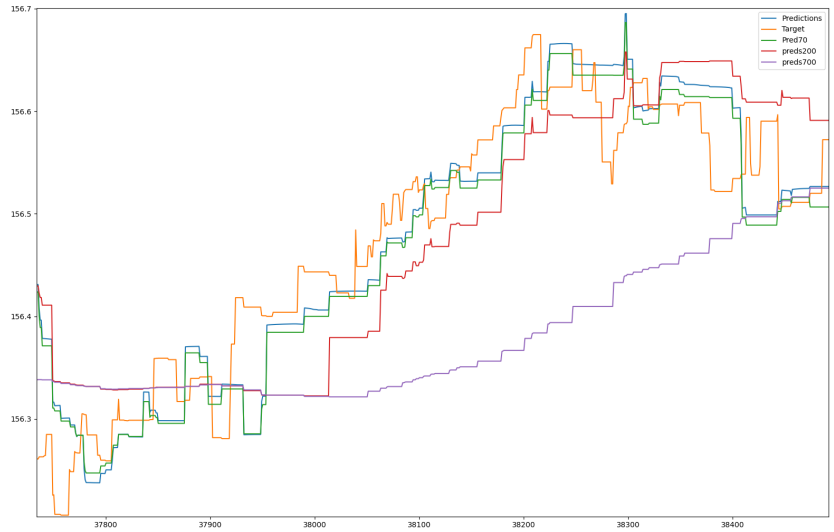


Figure 5.11: Graph shows stock price predictions for the distributed model *vol50_lr0.01_None_D*. Pred70, Preds200 and Preds700 refers to the prediction inputs used for the distributed model. Predictions for a 30 min Swedbank window.

Figure 5.10 shows a 3-hour prediction window for model *vol50_lr0.01_None_D*. The prediction for the distributed model follows the price movement of the tar-

get stock Swedbank. Taking a closer look, with a 30-minute window, in Figure 5.11 the distributed prediction follows the 70 market order single stock model *70_100e_lr0.0001_S* closely. However, it often differs in amplitude, scoring a better loss value than the single stock. For comparison the distributed model *rsi5_rsi30_vol100_vol50_ema30_ema15_lr0.001_Nordea70_D*, using Nordea price prediction in its input features, is shown in a 3-hour window in Figure 5.12 and a 30 minute window in Figure 5.13. The predictions from this model are very similar to *70_100e_lr0.0001_S*.

Figure 5.12: Graph shows stock price predictions for the distributed model *ema15_macd_rsi5_rsi30_vol100_vol50_lr0.001_NordeaPred70_D*.
Pred70, Preds200 and Preds700 refers to the prediction inputs used for the distributed model. Predictions for a 3 hour Swedbank window.

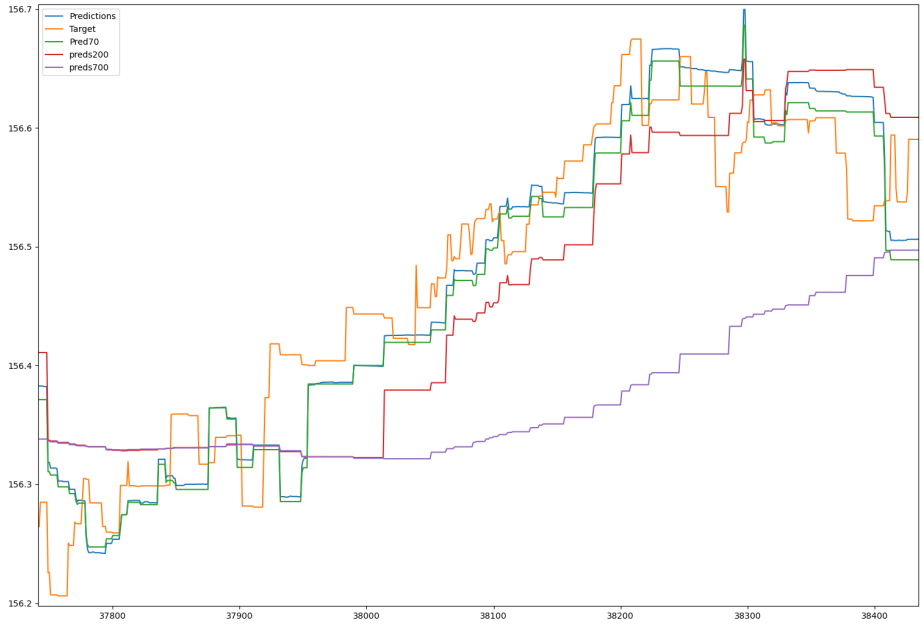


Figure 5.13: Graph shows stock price predictions for the distributed model *ema15_macd_rsi5_rsi30_vol100_vol50_lr0.001_NordeaPred70_D*. Pred70, Preds200 and Preds700 refers to the prediction inputs used for the distributed model. Predictions for a 30 min Swedbank window.

5.3.1 Discussion

The results presented in the section above show that the distributed models outperform the models *70_100e_lr0.0001_S*, *200_50e_lr0.0001_S*, and *700_100e_lr0.0001_S*. Since the distributed models perform better than any of the layer one models, the distributed model must utilize other accessible information. This information could be from financial indicators or extracted from several single stock predictors at once. For example, in Figure 5.12 the *700_100e_lr0.0001_S* lags behind, resulting in a smoother prediction curve, imitating a pseudo moving average. This imitation might increase the prediction capability of layer two models since it contributes to a larger historical aspect. Additionally, layer two models, which included several financial indicators, did not perform better than simpler models. This worse performance might be because of poor utilization of financial data or neural networks failing to find complex patterns between the financial indicators. Either way, the results for the layer two models do not show any specific correlation between any financial indicator and lower loss, possibly hitting that the performance gain of the layer two models came from comparing predictions from layer one models.

The distributed model uses predictions from pre-trained models using different window sizes. These pre-trained layer one models a large amount of data in order to

convergence, this is fine expect this training data cannot be reused for use with the distributed model, since this would entail that layer one models would predict on already seen data, thus falsely increasing performance. Because of this, the amount of data available for training the distributed system is the test data set for the layer one models, which is substantially smaller, possibly affecting performance for the distributed models.

Figure 5.11 and Figure 5.13 show the 30-minute windows for models *vol50_lr0.01_None_D* and *ema15_macd_rsi5_rsi30_vol100_vol50_lr0.001_NordeaPred70_D* respectively. The prediction graphs in theses figures are quiet similar, both resembling the graph for model *70_100e_lr0.0001_S* (pred70 in the Figures legend). This indicates that any influence financial indicators has on our models is marginal, making it hard to distinguish if financial indicators contributed to performance increase. This similarity could also be seen for all models in Table 5.6, graphs showing this can be found in Appendix I.

5.4 General remarks

This section describes patterns that could be seen generally when training and testing on different data.

5.4.1 Convergence Towards Average

Many of the trained models with the lower loss show that the training converges towards the mean of the window size data. Figure 5.14 indicates this, displaying the average and prediction overlap in many cases.

5.4.2 Offset in X and Y axis

Many of the trained models, either good or bad, often end up looking to be shifted in the x and y-axis. Figure 5.15 shows an example of this. We have not been able to draw any conclusion as to why this happens, but one theory about the shift in the x-axis could be that the neural network's next prediction could be the same as the latest price it got or the average price, which would end up looking like a reasonable shifted prediction.

5.4.3 Deep and Shallow Network performance

Several different networks were tested, but *The deep network* and *The shallow network* were the ones ending up used to measure performance. From testing, we saw that prediction performance benefited from *The deep network*. *The shallow network* and other shallow models failed to learn meaningful patterns in the data, regardless of the input vector size tested.

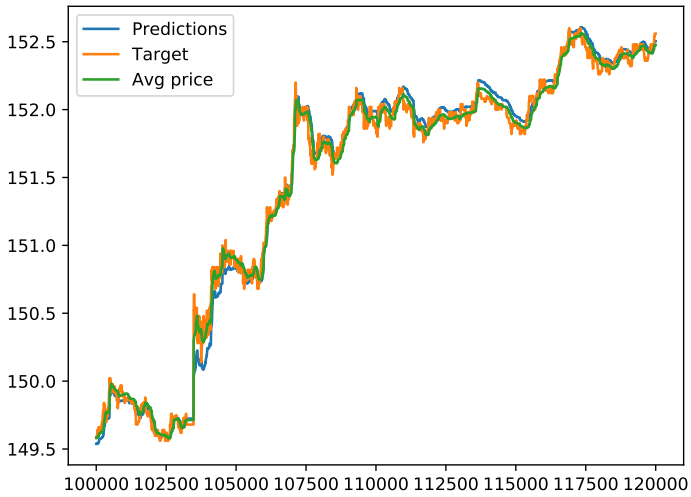


Figure 5.14: Price prediction result of a section of the test data for Swedbank_A, using model **ema_70_35E_30s_1e-06_time1**. X-axis is numbering of data-points from the start of the test set. Data collected from 08/25-2020 to 15/3-2021

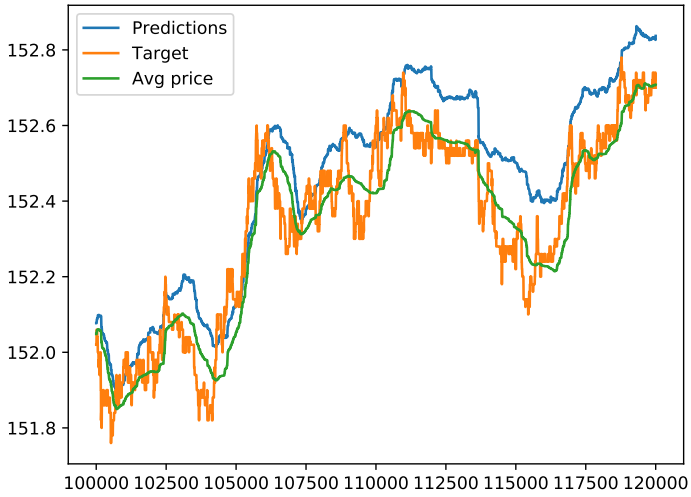


Figure 5.15: Price prediction result of a section of the test data for Swedbank_A, using model **price_200_5E_15s_1e-06_time1**. X-axis is numbering of data-points from the start of the test set. Data collected from 08/25-2020 to 15/3-2021

5.4.4 Network instability

When training *The deep network* with different combinations parameters, it was found that the results are volatile between runs, as Figure 5.16 shows. The diagram shows the difference in test loss between identical parameter settings and model architecture, training over the same data. Applying normalization reduces

this instability, as seen in Figure 5.17. Testing different activation functions showed additional improvements. Replacing Rectified linear unit (ReLU) with leaky ReLU proved to contribute to overall stability. However, the training remained in large unstable. Further attempts to reduce this instability, for example, using various weight initialization methods, had minimal impact.

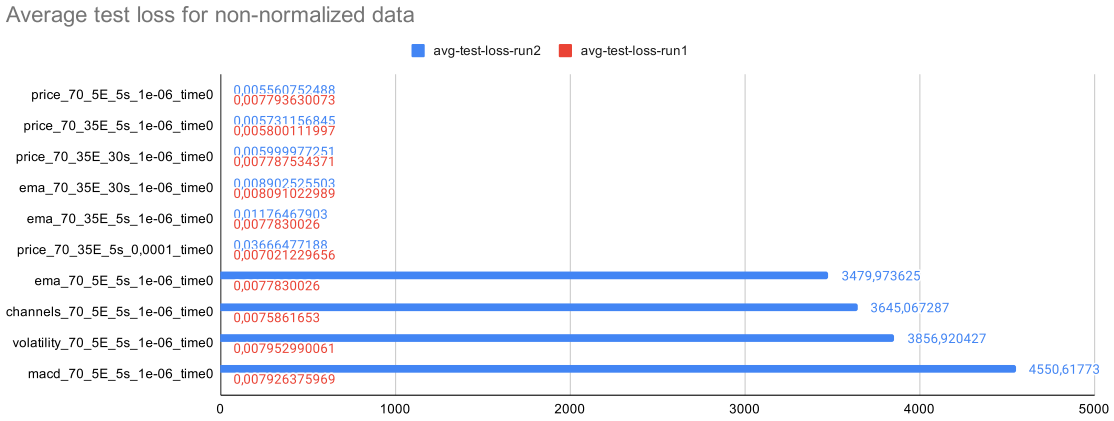


Figure 5.16: Average test loss for two independent runs using the same data. Normalization is not applied.

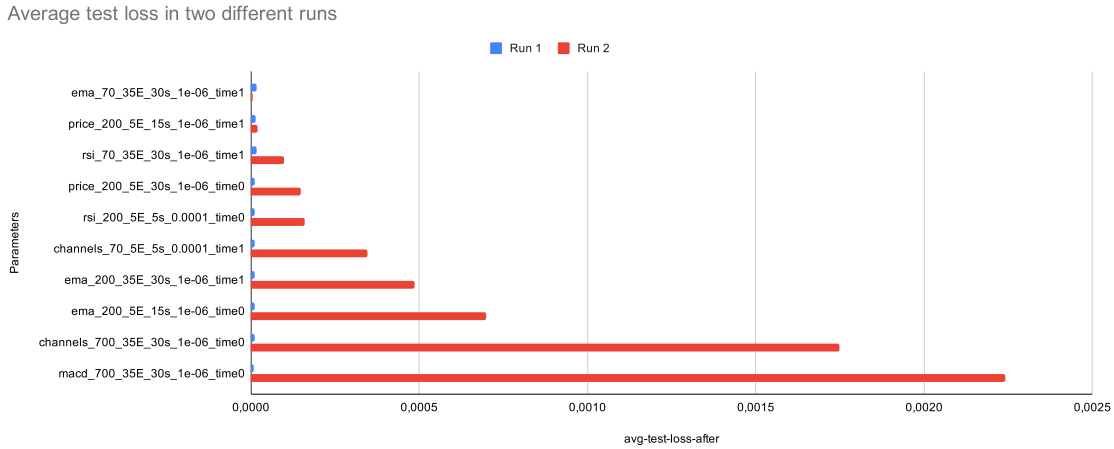


Figure 5.17: Average test loss for two independent runs using the same data. Normalization applied.

5.4.5 Financial Indicators Combined With Market Orders

When using a feed-forward neural network, the input size is fixed and needs to be set before the network can start. Since the input data is individual market orders, the chosen window size of prior market orders to include mainly decides this size. Other data that can affect the size are various financial indicators. These are in

some cases calculated on additional historical data than the market orders included in the current window. Thus they can give the model additional insight into the prior movement and market trends. Another data feature to include is the time any market order was placed. This data should, in theory, give the model the required information to adjust predictions according to market pressure and volatility. All these various data combinations contribute to a more extensive search plane.

From all the different parameters, it is challenging to distinguish valuable patterns from the input data choice. After running the same setup repeatedly, there is little difference between using different financial indicators or leaving them out entirely. Also, using market order publication time shows some benefit over excluding it, but with a tiny margin.

6
Conclusion

Predicting the stock market using market orders is hard, even when only predicting 30 seconds into the future. With different window sizes of 70, 200, and 700 recent market orders, it was difficult to find any clear patterns of which window size is the best. Stability of the predictions was another issue, which normalization and Leaky ReLU improved. Using financial indicators together with the window sizes showed no clear benefit, and the result looked somewhat random when looking at which combination of indicators was the best.

A new prediction from both outputs of different model predictions of different window sizes and financial indicators indicated a better result than using a single window-size model. The impact from other indicators was hard to measure, as the loss was not improved. Adding the output from the Nordea stock as an input when predicting Swedbank this way did not increase the performance of the prediction.

The prediction loss was measured against the loss from the latest 10-minute average and the loss of the latest market order. The prediction loss was, in the best cases, always better than the 10-minute average. However, the latest market order loss had a similar performance compared to the best non-distributed machine learning prediction. Only the distributed machine learning with financial indicators had a lower loss than the loss of the latest market order.

Creating more advanced predictions by combing outputs with financial indicators resulted in a lower loss. Running a distributed prediction with nine node-boxes in two layers created a worst-case delay of 0.07882 seconds. Such a slight delay relative to the 30-second prediction makes the distributed system viable for these kinds of predictions.

Using a window size of the latest market orders as input for artificial neural networks does not show a promising result. In most cases, for a 30-second prediction, a more accurate prediction is made by taking the latest available market order rather than using the created machine learning algorithm. However, when combining the prediction from several models and financial indicators, the prediction becomes more accurate than the latest market order when measured in MSE. Such a system of combining several predictions can also work in a real-time scenario with the distributed system implemented in this thesis.

6.1 Future work

This section describes further work that would be a suitable expansion upon this thesis. Some future work focuses on other techniques, while some focus on expanding the current work.

6.1.1 Long Short-Term Memory and Transformers

Long Short-Term Memory (LSTM) is a recurrent neural network machine learning algorithm. LSTM's architecture is made to be able to remember sequences of data, such as the history of the stock market [15].

Transformers is a machine learning architecture for predicting sequential data. Using an encoding decoding format, transformers lend themselves well for natural language processing (NLP) problems. Compared to RNN networks, transformers use no "recurring" part but only uses attention, creating connections with the past. Because of this transformers do not suffer from the vanishing gradient problem, as in the case for RNN or LSTMs. Furthermore, transformers sequentially encode the inputs before processing, thus eliminating the need to process inputs sequentially, allowing for easy parallel computations [46].

Both LSTM and Transformers have the benefit of remembering sequences of data. As this thesis's stock market predictions are based on the sequential history of market orders, LSTM and Transformers could give better results than the current machine learning algorithm.

6.1.2 Train model with data from several stocks

In this thesis, there was a combination of two stocks trained together, Swedbank and Nordea. Expanding this to more stocks from a sector, such as banking, mining, or telecommunication, could potentially give promising results.

6.1.3 Volume indicators

Currently, the volumes of trading are not taken into account. Adding an indicator for the volume could improve the machine learning performance, as the volume gives more information about what is happening in the stock. In this thesis, it is impossible to know if a price increase is from someone buying a few stocks or someone buying several millions of stocks. Some interesting indicators to look into could be *On-Balance Volume* [13] and *Klinger Oscillator* [27].

6.1.4 Train using node-boxes

As the node-boxes are currently not made to be able to train the machine learning in them, they have to be manually trained one by one. They also need first to be trained in the first layer to give good predictions to layer two. This process is

pretty time-consuming and makes it challenging to train many different node-box configurations. Implementing a way to train directly in the node-box configuration could make it easier to test different configurations and also possibly train faster, as it is distributed.

6.1.5 Optimize node-box code

The code for the node boxes is written in python. The code is not optimized for parallelism, and this could be a possible improvement. As future work, the code could be rewritten in a low-level language such as c or c++, focusing on parallelizing as much as possible, for example, the smart-sync.

6.1.6 Classification predictor

Instead of predicting a future price, a classification could instead be done to predict whether the price would go up, down, or stay the same. Such a classification has been done before by [5, 34] with some success, but it has not been tested directly with market orders, as in this thesis.

Bibliography

[1] Manish Agrawal, Asif Ullah Khan, and Piyush Kumar Shukla. "Stock price prediction using technical indicators: A predictive model using optimal deep learning". In: *Learning* 6.2 (2019), p. 7.

[2] Kamal Ahmed et al. "Multi-model ensemble predictions of precipitation and temperature using machine learning algorithms". In: *Atmospheric Research* 236 (2020), p. 104806.

[3] Yang Bing, Jian Kun Hao, and Si Chang Zhang. "Stock market prediction using artificial neural networks". In: *Advanced Engineering Forum*. Vol. 6. Trans Tech Publ. 2012, pp. 1055–1060.

[4] James Chen. *Donchian Channels Definition*. https://www.investopedia.com/terms/d/donchianchannels.asp (visited: 2021-03-22). 2021.

[5] Eunsuk Chong, Chulwoo Han, and Frank C. Park" network Covariance estimation. "Deep learning networks for stock market analysis and prediction: Methodology, data representations, and case studies". In: *Expert Systems with Applications* 83 (2017), pp. 187–205. ISSN: 0957-4174.

[6] Terence Tai-Leung Chong and Wing-Kam Ng. "Technical analysis and the London stock exchange: testing the MACD and RSI rules using the FT30". In: *Applied Economics Letters* 15.14 (2008), pp. 1111–1114. DOI: 10.1080/13504850600993598. eprint: https://doi.org/10.1080/13504850600993598. URL: https://doi.org/10.1080/13504850600993598.

[7] Callum Cliffe. *A trader's guide to Donchian channels*. https://www.ig.com/uk/trading-strategies/a-trader-s-guide-to-donchian-channels-200218 (visited: 2021-03-22). 2020.

[8] Matthew Dixon, Diego Klabjan, and Jin Hoon Bang. "Classification-based financial markets prediction using deep neural networks". In: *Algorithmic Finance* 6.3-4 (2017), pp. 67–77.

[9] Adrian A Drăgulescu and Victor M Yakovenko. "Probability distribution of returns in the Heston model with stochastic volatility". In: *Quantitative finance* 2.6 (2002), pp. 443–453.

[10] Louis H. Ederington and Wei Guan. "Measuring Historical Volatility". In: *Journal of Applied Finance* 16 (2006).

[11] David Enke and Suraphan Thawornwong. "The use of data mining and neural networks for forecasting stock market returns". In: *Expert Systems with applications* 29.4 (2005), pp. 927–940.

[12] Markus Glaser and Martin Weber. "Momentum and Turnover: Evidence from the German Stock Market". In: *Schmalenbach Business Review* 55.2 (Apr.

2003), pp. 108–135. ISSN: 2194-072X. DOI: 10.1007/BF03396669. URL: https://doi.org/10.1007/BF03396669.

[13] J.E. Granville. *Granville's New Key to Stock Market Profits*. Papamoa Press, 2018. ISBN: 9781789126037. URL: https://books.google.se/books?id=21ukDwAAQBAJ.

[14] Jeff Hajewski and Suely Oliveira. "Distributed SmSVM Ensemble Learning". In: *Recent Advances in Big Data and Deep Learning*. Ed. by Luca Oneto et al. Cham: Springer International Publishing, 2020, pp. 7–16. ISBN: 978-3-030-16841-4.

[15] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[16] Ehsan Hoseinzade and Saman Haratizadeh. "CNNpred: CNN-based stock market prediction using a diverse set of variables". In: *Expert Systems with Applications* 129 (2019), pp. 273–285. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2019.03.029. URL: https://www.sciencedirect.com/science/article/pii/S0957417419301915.

[17] Narasimhan Jegadeesh and Sheridan Titman. "Returns to buying winners and selling losers: Implications for stock market efficiency". In: *The Journal of finance* 48.1 (1993), pp. 65–91.

[18] Kyoung-jae Kim and Won Boo Lee. "Stock market prediction using artificial neural networks with optimal feature transformation". In: *Neural computing & applications* 13.3 (2004), pp. 255–260.

[19] T. Kimoto et al. "Stock market prediction system with modular neural networks". In: *1990 IJCNN International Joint Conference on Neural Networks*. 1990, 1–6 vol.1. DOI: 10.1109/IJCNN.1990.137535.

[20] Douglas M. Kline and Victor L. Berardi. "Revisiting squared-error and cross-entropy functions for training neural network classifiers". In: *Neural Computing & Applications* 14.4 (Dec. 2005), pp. 310–318. ISSN: 1433-3058. DOI: 10.1007/s00521-005-0467-y. URL: https://doi.org/10.1007/s00521-005-0467-y.

[21] Mu Li et al. "Scaling Distributed Machine Learning with the Parameter Server". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 583–598. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[22] Lu Lu. "Dying ReLU and Initialization: Theory and Numerical Examples". In: *Communications in Computational Physics* 28.5 (June 2020), pp. 1671–1706. ISSN: 1991-7120. DOI: 10.4208/cicp.oa-2020-0165. URL: http://dx.doi.org/10.4208/cicp.OA-2020-0165.

[23] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*. 2013.

[24] James D MacBeth and Larry J Merville. “An empirical examination of the Black-Scholes call option pricing model”. In: *The journal of finance* 34.5 (1979), pp. 1173–1186.

[25] Dilip B Madan, Peter P Carr, and Eric C Chang. “The variance gamma process and option pricing”. In: *Review of Finance* 2.1 (1998), pp. 79–105.

[26] Burton Malkiel. “The Efficient Market Hypothesis and Its Critics”. In: *Journal of Economic Perspectives* 17 (Feb. 2003), pp. 59–82. DOI: 10.1257/089533003321164958.

[27] Cory Mitchell. *Klinger Oscillator Definition*. Aug. 2020. URL: https://www.investopedia.com/terms/k/klingeroscillator.asp.

[28] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[29] Mahdi Pakdaman Naeini, Hamidreza Taremian, and Homa Baradaran Hashemi. “Stock market value prediction using neural networks”. In: *2010 international conference on computer information systems and industrial management applications (CISIM)*. IEEE. 2010, pp. 132–136.

[30] *Nasdaq Nordiq*. http://www.nasdaqomxnordic.com/.

[31] Aparna Nayak, MM Manohara Pai, and Radhika M Pai. “Prediction models for Indian stock market”. In: *Procedia Computer Science* 89 (2016), pp. 441–449.

[32] *Normalization*. https://www.codecademy.com/articles/normalization (visited: 2021-04-15).

[33] Xiongwen Pang et al. “An innovative neural network approach for stock market prediction”. In: *The Journal of Supercomputing* 76.3 (2020), pp. 2098–2118.

[34] Jigar Patel et al. “Predicting stock market index using fusion of machine learning techniques”. In: *Expert Systems with Applications* 42.4 (2015), pp. 2162–2172. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2014.10.031. URL: https://www.sciencedirect.com/science/article/pii/S0957417414006551.

[35] PyTorch. *PyTorch Docs*. Mar. 2021. URL: https://pytorch.org/docs/stable/index.html.

[36] Bo Qian and Khaled Rasheed. “Stock market prediction with multiple classifiers”. In: *Applied Intelligence* 26.1 (2007), pp. 25–33.

[37] R. Rosillo, D. de la Fuente, and J. A. L. Brugos. “Technical analysis and the Spanish stock exchange: testing the RSI, MACD, momentum and stochastic rules using Spanish market companies”. In: *Applied Economics* 45.12 (2013), pp. 1541–1550. DOI: 10.1080/00036846.2011.631894. eprint: https://doi.org/10.1080/00036846.2011.631894. URL: https://doi.org/10.1080/00036846.2011.631894.

[38] U.S. Securities and Exchange Commission. *Insider Trading*. https://www.investor.gov/introduction-investing/investing-basics/glossary/insider-trading (visited: 2021-03-22).

[39] U.S. Securities and Exchange Commission. *Pump and Dump Schemes*. https://www.investor.gov/introduction-investing/investing-basics/glossary/pump-and-dump-schemes (visited: 2021-03-22).

[40] Sreelekshmy Selvin et al. “Stock price prediction using LSTM, RNN and CNN-sliding window model”. In: *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. 2017, pp. 1643–1647. DOI: 10.1109/ICACCI.2017.8126078.

[41] Justin A Sirignano. “Deep learning for limit order books”. In: *Quantitative Finance* 19.4 (2019), pp. 549–570.

[42] Adrian Ţăran-Moroşan. “The relative strength index revisited”. In: *African Journal of Business Management* 5.14 (2011), pp. 5855–5862.

[43] M.T. Tham. *Exponentially Weighted Moving Average Filter*. https://web.archive.org/web/20100329135531/http://lorien.ncl.ac.uk/ming/filter/filewma.htm (visited: 2021-03-12).

[44] Allan Timmermann and Clive WJ Granger. “Efficient market hypothesis and forecasting”. In: *International Journal of forecasting* 20.1 (2004), pp. 15–27.

[45] Avraam Tsantekidis et al. “Forecasting Stock Prices from the Limit Order Book Using Convolutional Neural Networks”. In: *2017 IEEE 19th Conference on Business Informatics (CBI)*. Vol. 01. 2017, pp. 7–12. DOI: 10.1109/CBI.2017.23.

[46] Ashish Vaswani et al. “Attention is all you need”. In: *arXiv preprint arXiv: 1706.03762* (2017).

[47] *Volume Statistics | Nasdaq, Inc.* en. URL: https://ir.nasdaq.com/financials/volume-statistics (visited on 11/26/2020).

[48] Qi Wang et al. “A Comprehensive Survey of Loss Functions in Machine Learning”. In: *Annals of Data Science* (Apr. 2020). DOI: 10.1007/s40745-020-00253-5.

[49] C. Willmott and K Matsuura. “Advantages of the Mean Absolute Error (MAE) over the Root Mean Square Error (RMSE) in Assessing Average Model Performance”. In: *Climate Research* 30 (Dec. 2005), p. 79. DOI: 10.3354/cr030079.

[50] M. Wu and X. Diao. “Technical analysis of three stock oscillators testing MACD, RSI and KDJ rules in SH SZ stock markets”. In: *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*. Vol. 01. 2015, pp. 320–323. DOI: 10.1109/ICCSNT.2015.7490760.

[51] Bing Xu et al. *Empirical Evaluation of Rectified Activations in Convolutional Network*. 2015. arXiv: 1505.00853 [cs.LG].

[52] Seyed Hadi Mir Yazdi and Ziba Habibi Lashkari. “Technical analysis of Forex by MACD Indicator”. In: *International Journal of Humanities and Management Sciences (IJHMS)* 1.2 (2013), pp. 159–165.

[53] Ban Zheng, Eric Moulines, and Frédéric Abergel. *Price Jump Prediction in Limit Order Book*. 2012. arXiv: 1204.1381 [q-fin.TR].

A

Appendix 1

Figure A.1: Prediction of Nordea stock with 70 market orders as input, using model *70_100e_lr0.0001_N*. The graph is shown over a 3 hour window.

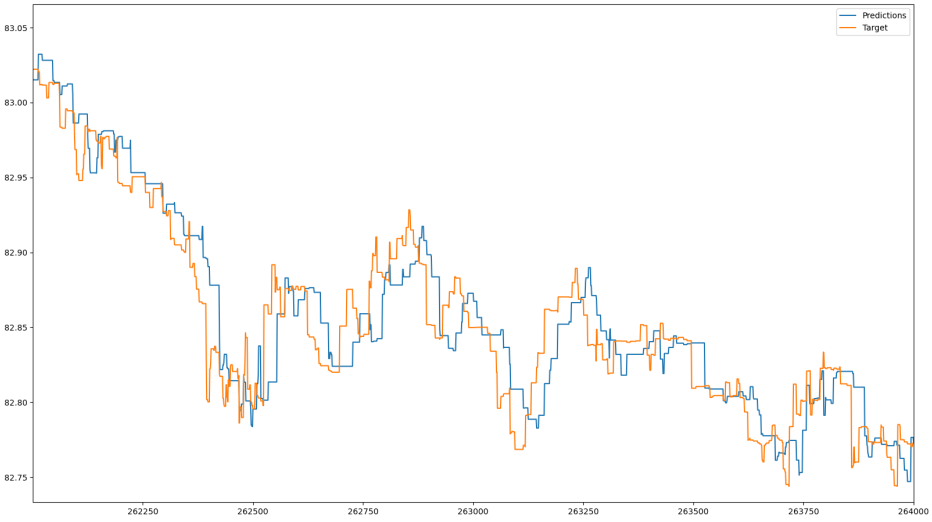


Figure A.2: Prediction of Nordea stock with 70 market orders as input, using model *70_100e_lr0.0001_N*. The graph is shown over a 30 minute window.

Figure A.3: Prediction of Nordea stock with 200 market orders as input, using model *200_100e_Ir0.0001_N*. The graph is shown over a 3 hour window.

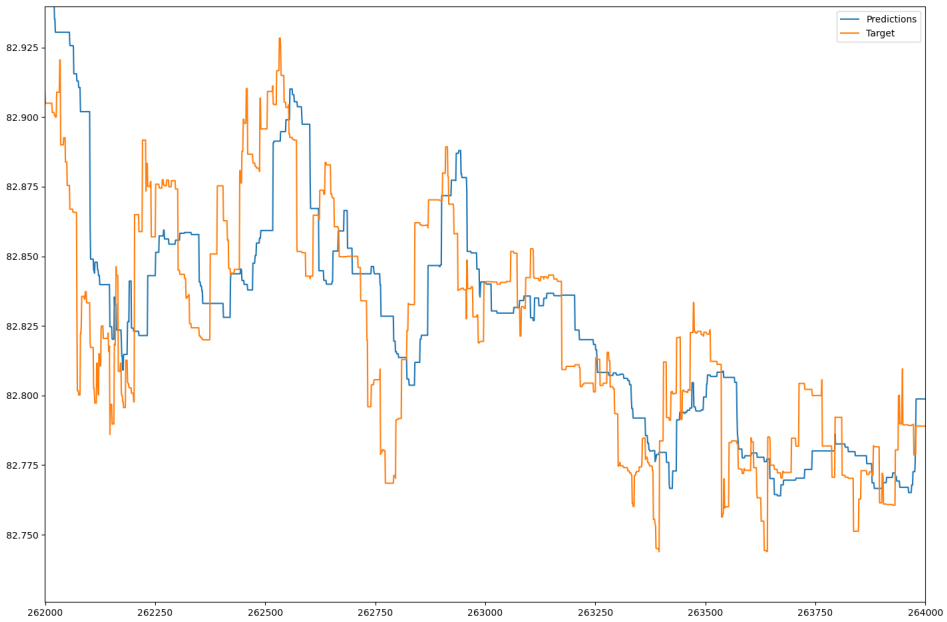


Figure A.4: Prediction of Nordea stock with 200 market orders as input, using model *200_100e_Ir0.0001_N*. The graph is shown over a 30 minute window.

Figure A.5: Prediction of Nordea stock with 700 market orders as input, using model *700_30e_lr1e-05_N*. The graph is shown over a 3 hour window.

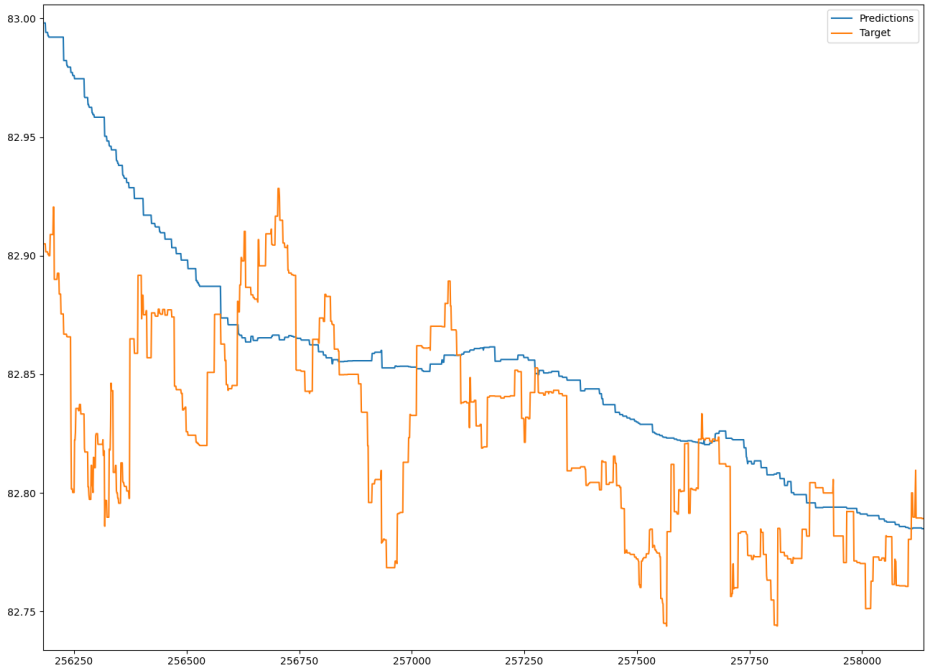


Figure A.6: Prediction of Nordea stock with 700 market orders as input, using model *700_30e_lr1e-05_N*. The graph is shown over a 30 minute window.

Figure A.7: Prediction of Swedbank stock with 70 market orders as input, using model *70_100e_lr0.0001_S*. The graph is shown over a 3 hour window.

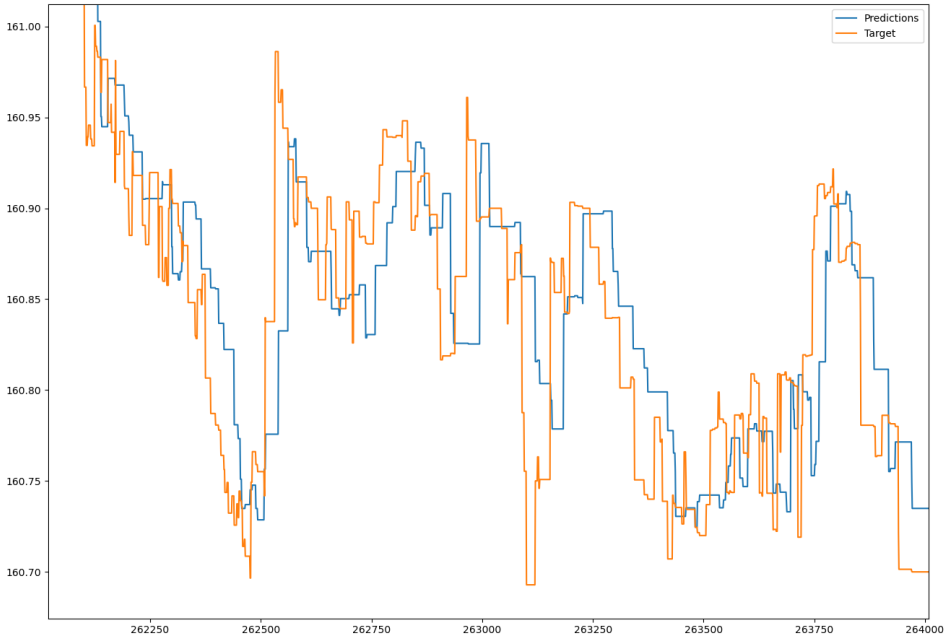


Figure A.8: Prediction of Swedbank stock with 70 market orders as input, using model *70_100e_lr0.0001_S*. The graph is shown over a 30 minute window.

Figure A.9: Prediction of Swedbank stock with 200 market orders as input, using model *200_50e_lr0.0001_S*. The graph is shown over a 3 hour window.

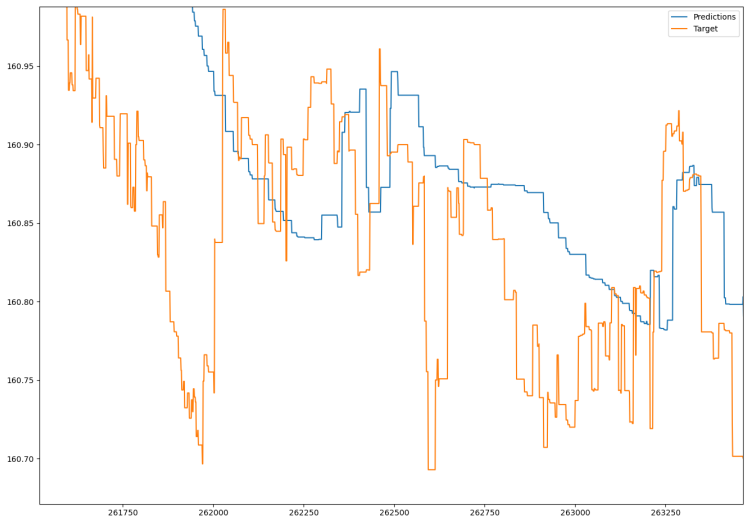


Figure A.10: Prediction of Swedbank stock with 200 market orders as input, using model *200_50e_lr0.0001_S*. The graph is shown over a 30 minute window.

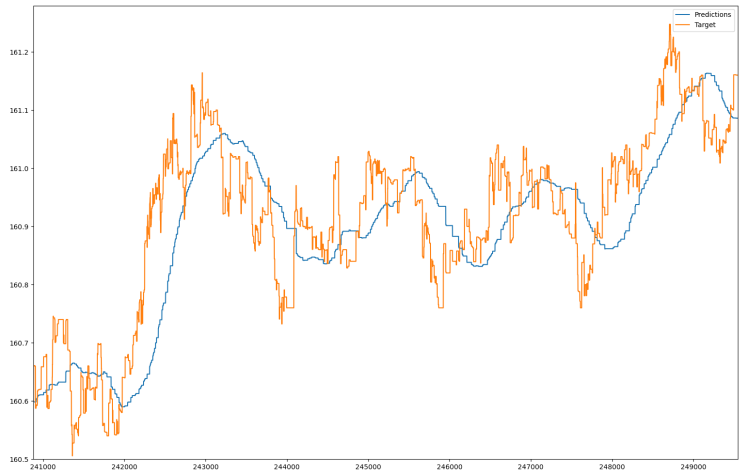


Figure A.11: Prediction of Swedbank stock with 700 market orders as input, using model *700_100e_lr0.0001_S*. The graph is shown over a 3 hour window.

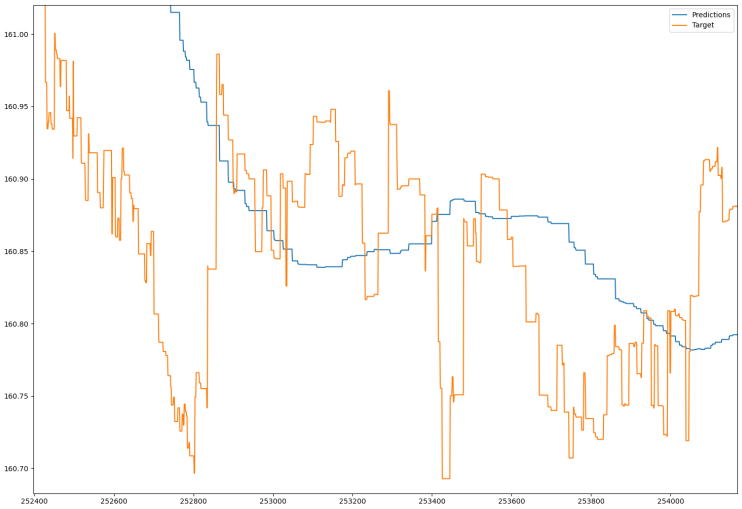


Figure A.12: Prediction of Swedbank stock with 700 market orders as input, using model *700_100e_lr0.0001_S*. The graph is shown over a 30 minute window.

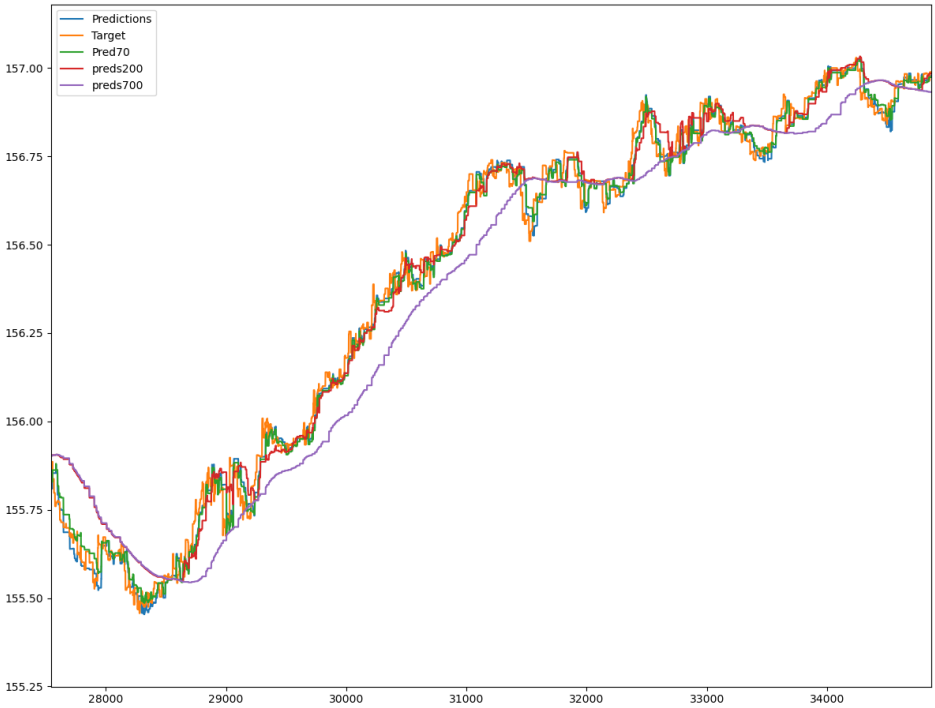


Figure A.13: Prediction of Swedbank stock using the distributed model *rsi30_lr0.01_None_D*. The graph is shown over a 3 hour window.

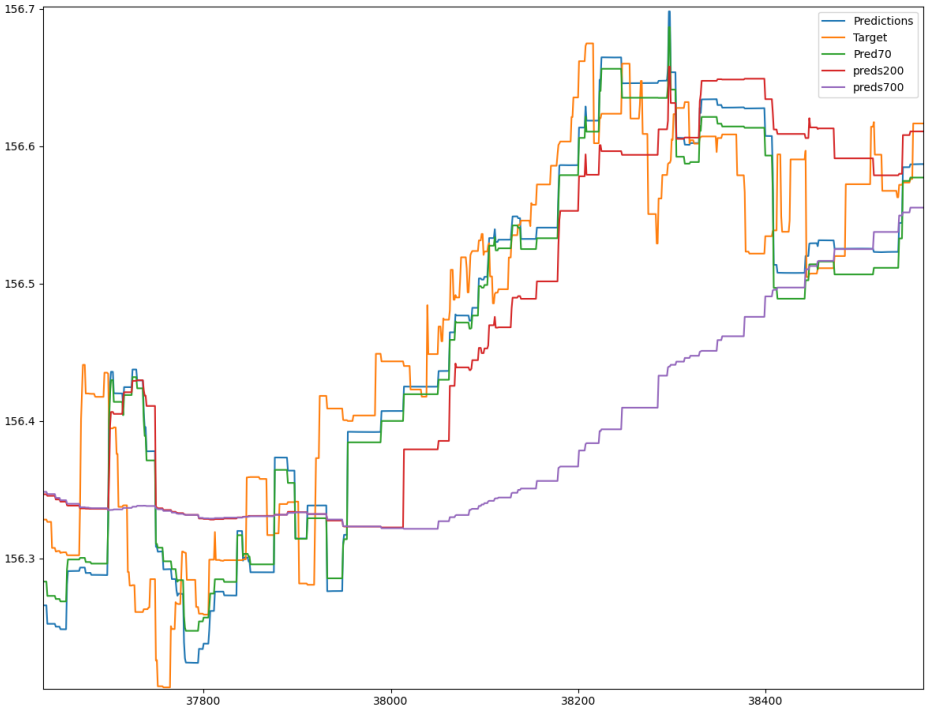


Figure A.14: Prediction of Swedbank stock using the distributed model *rsi30_lr0.01_None_D*. The graph is shown over a 30 minute window.