An Artificial Neural Network Approach to Algorithmic Trading

Timmie Bengtsson

January 2023

Degree Project in Mathematical Statistics

Lund University
Faculty of Engineering
Centre for Mathematical Sciences
Mathematical Statistics

Popular Science Summary

An Artificial Neural Network Approach to Algorithmic Trading

Timmie Bengtsson

TODO————————————————————

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Abstract

The field of machine learning has advanced significantly in the last decades, and at the same time, computational power has improved to the point where it is now accessible to train large machine learning models like artificial neural networks. This has lead to an rise in use of these models within the financial sector, with some firms leveraging the models to assist them with their investment decisions. Using neural networks, or machine learning models in general, for investing refers to investment strategies constructed by, at least partially, training algorithms on historical data to identify patterns that may repeat in the future. The idea behind this being a successful approach for trading financial markets is that historical data contains structure that will be repeated again in the future. In the context of asset prices, this means that previous price developments of an asset holds valuable information that can be used to predict future price developments, challenging traditional market theories, such as the efficient market hypothesis. Despite this, the price patterns have proven reliable enough to allow multiple players to reap huge financial gains and develop into organizations with headcount in the thousands; focused solely on finding and trading these pricing patterns. The goal of this thesis is to construct and assess artificial neural network models for use in trading algorithms. Given historical returns, the models are trained to forecast the direction of asset price returns the following day. The predicted return directions are then inserted into a trading algorithm that calculates daily portfolio values resulting from using the models predictions. These models are then benchmarked against naive trading strategies and the underlying asset itself. Benchmarks values used to evaluate the models are the Sharpe ratio, the Sortino ratio, gross return and maximal drawdown. The underlying assets are government bond contracts at the request of Handelsbanken Fonder.

The results do not indicate any improvement in return estimation compared to the naive benchmark models. The model performing best on average is INSERT which had the highest average Sharpe ratio of INSERT, Sortino ratio of INSERT and gross returns of INSERT.

This thesis was written in collaboration with Handelsbanken Fonder as the concluding part of a master's degree in Financial Engineering at Lund University.

Keywords: Financial Markets, Machine Learning, Long Short-Term Memory, Gated Recurrent Unit, Recurrent Neural Networks, Time Series Analysis, Algorithmic Trading.

Acknowledgements

I would like to thank my assistant supervisor Carl Hvarfner for his contributions and for being a great supervisor to work with; always flexible and quick to answer any questions. Highly appreciated Carl! Erik Lindström also deserves a recognition for his high value inputs, thank you for your expertise Erik. I would also like to thank Handelsbanken Fonder for their support, ideas and for providing the data used in the thesis. For those interested in doing a thesis within quantitative finance, I absolutely recommend Handelsbanken Fonder!

Contents

List of Figures

List of Tables

1

Introduction

The branch of applied mathematics called quantitative finance is devoted to the mathematical modeling of financial markets, and predicting future asset prices-—or their direction—-relates to a number of problems in this area. A motivator for this type of research is increasing efficiency in capital markets (Hübner, 29-30 November 2010). The ones succeeding in increasing the efficiency of the markets, meaning that they are decreasing or removing mispricings, will in turn be profiting from this since profiting is inherently the result of taking actions that corrects the market. With profits being the result of increasing the efficiency in markets, and profits increasing with the level of success, the incentives to participate in trading financial markets are quite obvious.

Hedge funds are one of the market participants frequently associated with quantitative finance, with the funds heavily focused on quantitative strategies often referred to as quant funds. This means that the funds trading choices are based on algorithmic, systematic procedures (Securities and Commission). If a fund can produce predictions of asset movements that consistently are good enough, then, trading algorithms leveraging these predictions can generate high risk adjusted returns for the fund. One of the most well known such funds are Renaissance Technologies, which bolstered no less than a 66% annualized return during the 30-year span from 1988 to 2018 (Zuckerman, 2019a). A non-quantitative hedge fund, on the other hand, normally bases its trading strategy on fundamental analysis, and instead the fund managers are in charge of the trading decisions.

According to the Efficient Market Hypothesis (EMH), asset prices in capital markets accurately reflect all of the information that is currently available, and new information will be immediately factored into the price. This implies that asset price development is inherently unpredictable it is therefore impossible to use any information available to forecast returns. EMH proponents assert that attempts by researchers and practitioners to develop predictive models are meaningless; the only avenue towards generating excess returns are through chance or exposure to riskier assets (Fama, 1970), (Fama, 1965). However, EMH has been challenged multiple times. (Grossman and Stiglitz, 1980) showed that if knowledge is expensive to collect, the investor will be compensated in accordance with that cost, something that (Ippolito, 1989) confirmed later on. It also appears that stocks with low price to earning (P/E) ratios outperform those with high P/E ratios (Dreman and Berry, 1995). According to Malkiel (2003), an increasing number of financial economists and statisticians disagree with the EMH, they contend that both technical and fundamental information has predictive value. Malkiel (2003) does however affirms his trust in markets being efficient in the long run, despite acknowledging the existence of short-term predictive patterns. Alike Malkiel (2003), Timmermann and Granger (2004) also notes that there are short-term predictive structures in financial markets but holds the belief that there are no long-term forecasting patterns, this because they believe that the patterns will be crowded out if made public. The self-destruction of patterns that Timmermann and Granger (2004) describes have consequences, one being that financial forecasting researchers may be unwilling to share successful predictive models, instead they may choose to sell the models or keep them for private use.

Findings exposing EMH's shortcomings are obviously welcomed by hedge funds, as the possibility to generate excess returns underpins the industry. Though one might argue that the success, over long stretches of time, for some hedge funds, constitutes evidence itself against the EMH. This continuous long-term success is no small feat considering that as predictive models–or the structures they exploit–become known and their predictive power disappears, the new structures emerging from this, and the ones still remaining, will be increasingly complex; implying that trading markets successfully gets harder over time. A possible real world hint of this is that, in 2011, Goldman Sachs closed down their well-known hedge fund relying on computer-driven trading (Lauren Tara LaCapra, 2011), indicating that even some of the biggest institutional players were struggling. Hence, it is obvious that the increasing complexity of patterns drive the need for using models capable of extracting these otherwise almost unrecognizable patters. (Huang et al., 2020) demonstrates that artificial neural networks (ANNs) have been widely used in the prediction of stock markets, exchange rates but also for portfolio management, macroeconomic forecasting, and default risks.

The emergence of machine learning techniques in parallel to computing power during the last decades has showcased the immense potential of these models for a number of problems; including prediction. Google's AlphaFold (Jumper et al., 2021) which predicts 3D models of protein structures is a renowned example of this. Although these later developments might have promoted use of machine learning in finance, machine learning–especially ANNs–was already a fast growing area within financial forecasting during the 1990s (Zhang et al., 1998). Widrow et al. (1994) reported in 1994 that multiple 1990s financial sector giants such as Salomon Brothers, Lehman Brothers, Citibank and Merrill Lynch, employed ANNs for financial forecasting and portfolio management; with Citibank claiming that their neural network models yielded 25% yearly returns trading currency markets. Renaissance Technologies allegedly used machine learning methods for bet sizing in 1992, and later developed machine learning systems that ran on their own (Zuckerman, 2019b); indicating, when considering their returns, that machine learning methods are indeed effective for predicting asset prices and assisting in trading algorithms. BlackRock, the world's largest asset manager as defined by assets under management, confirms this by stating that they believe machine learning and big data technologies being able to boost investors ability to generate alpha (Raffaele Savi, 2015).

Unfortunately, the strategies used by Renaissance and hedge funds alike are normally kept secret, hence limiting insight into the research and working strategies of these funds. Researches in academia, on the other hand, are incentivised to publish, and from this research a number of insights can be be gained. For example, Blume et al. (1994) shows how information on historical stock prices together with trading volume can be informative of future stock price movements. Enke and Thawornwong (2005) uses feedforward neural networks and comes to the conclusion that, given the right inputs, a neural network model outperforms both linear regression model and buy-and-hold strategies. For derivative assets, pricing formulas derived by using neural networks may be more accurate than traditional methods under certain conditions Hutchinson et al. (1994). When comparing state-of-the-art deep learning models, Livieris et al. (2020) analysis indicates that long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997) models are the most effective for predicting gold prices. Lo et al. (2000) Used a non-parametric kernel regression approach for pattern recognition on a large number of US stocks, spanning the 31 years from 1962 to 1996, and showed that several technical indicators did provide predictive properties during this period. When predicting the direction of change of the Taiwan Stock market index (Chen et al., 2003) found that the best performing model was a probabilistic neural network (PNN) trained on historical data. Chatigny et al. (2021) employs attention-guided (Vaswani et al., 2017) deep learning to identify the most influential firm characteristics and uses the results to construct a mean-variance optimized portfolio that ends up performing well compared to existing models. In 2020, (Zhang et al., 2020) showed that reinforcement learning algorithms designed for trading futures contracts could deliver positive profits despite heavy transaction costs on 50 very liquid contracts between 2011 and 2019. A finding highlighting that it has in recent years, despite the markets now being so competitive, been possible to design profitable algorithmic trading strategies.

Attempts at modelling markets has also been approached from a visually inspired avenue. Convolution neural networks (CNN) (Fukushima, 1988), normally used for image recognition tasks, was used to provide visual representations of stock market data features and achieved state-of-the-art performance in the domain of using neural networks for technical forecasting Ghoshal and Roberts (2020). Again using a CNN approach, but this time combined with LSTM, Zhang et al. (2019) could predict predict price movements from limit order books (LOB) data of cash equities. Their network outperformed all existing state-of-the-art algorithms and produced stable out-of-sample prediction accuracy for a variety of instruments on London Stock Exchange (LSE). These predictions generated statistically significant profits; indicating that the model is capable of extracting universal features.

1.1 Objective and Scope

The objective of this thesis is to develop and assess artificial neural network models that predict returns of financial time series and then use these predictions in trading algorithms. The scope will be limited to training and evaluating the models to forecast and trade the five credit assets supplied by Handelsbanken Fonder.

2

Theory

2.1 Finance

2.1.1 Long and short positions

An investor who buys an asset is said to take a long position in that asset. If the asset price increases over the holding period, the investor's investment will increase in value. If the asset price decreases, the investor's investment will instead decrease in value. The investor's returns on invested capital will correspond to the change in the asset's value over the holding period.

A short position is in a sense the opposite of a long position. In practice, a short position is commonly associated with a net short position instead of just selling an existing holding, meaning that the investor owns a negative amount of some asset. An investor achieves this by, for example, borrowing an asset from another investor and then selling that asset.

2.1.2 Trading bots and algorithmic trading

Someone who, usually as their full-time profession, enters shorter-term positions in financial instruments could be called a trader. A successful trader, loosely defined as a trader who achieves a positive return on capital over time from their trading activity, takes decisions based on data and execute the trades they believe will be profitable. The data analyzed in order to reach trading decisions varies, but almost, if not all, traders analyze historical time series data of assets. This process of roughly: (1) observe data, (2) analyze the data, (3) form a decision, and finally (4) execute on the decision, could be replicated by a computer. If implemented by code this is what normally is referred to as a trading bot, when these trading bots then are deployed for trading on an exchange, it is referred to as ”algorithmic trading”, indicating the fact that it is not a human who trades but rather an algorithm.

2.1.3 Performance metrics

Sharpe-ratio

A common way to benchmark financial performance of assets, or more commonly groups of assets in a portfolio, is by comparing Sharpe-ratios (SR) (Sharpe, 1966).

The Sharpe-ratio is a way to quantify financial performance with a so called risk-adjusted return, hence it will be suitable for comparisons of the different trading strategies in this thesis. The Sharpe-ratio is defined as follows,

$$\text{Sharpe-ratio} = \frac{r_p - r_f}{\sigma_p}$$

where r_p is the average yearly return, r_f the risk-free rate and σ_p the yearly volatility. This formula is only valid for annualized returns and volatilities, however, those values are sometimes not available. Thus, for shorter time frames, the ratio is usually multiplied by the square root of the number of periods in a year, like $sqrt(252)$ for daily returns and volatilities. Unfortunately, this simple method has been shown to produce inaccurate estimates (Lo, 2002), see 3.5 for how this issue was dealt with.

Sortino-ratio

Sortino and Van Der Meer (1991) developed the Sortino ratio which is a measure comparable to the Sharpe-ratio but that differs in one critical way: it only takes downside risk into consideration. The ratio's justification is that an investor wouldn't mind if a portfolio's return showed significant fluctuation exclusively on the upside. Even though it exhibits significant fluctuation, a portfolio that yields 3% the first month, 50% the next month and finally 10% would likely satisfy most investors. Instead, what investors are typically more concerned about is downside volatility. The Sortino-ratio is calculated as follows,

$$\text{Sortino Ratio} = \frac{r_p - r_f}{\sigma_{p-negative}}$$

where

$$\sigma_{\text{p-negative}} = \left(\frac{1}{n} \sum_{k=1}^{n} \min(0, r_k)^2 \right)^{1/2}$$

where n is the number of periods and r_k the return at period k

Gross return

Gross return is the total return achieved over a period, for a time period $t = t_0 \dots t_n$ it is found by,

$$\text{Gross return}(t_n) = \frac{y_{t_n} - y_{t_0}}{y_{t_0}}$$

Maximal-drawdown

Maximal drawdown is the biggest loss as counted from the last peak. Let A_t be the value of an asset at time t, then its running maximum M_t is given by,

$$M_t = \max_{u \in [0,t]} A_u.$$

Maximum drawdown, MDD_t, is then defined as (Pospisil and Vecer, 2008) the largest drop in the asset price from the running maximum up to time t,

$$MDD_t = \max_{u \in [0,t]} (M_u - A_u)$$

Statistical measures

Previously mentioned measures evaluates the models performance from a financial perspective. These measures are not optimal from a scientific perspective due to the potentially large effects single returns can have on the performance, see 3.5.2. Therefore, some statistical measures are introduced as well. Lets start with the confusion matrix and its components; true negatives, true positives, false negatives and false positives.

		Prediction		
		pos	**neg**	**total**
Actual	**pos**	True Positive	False Negative	P
	neg	False Positive	True Negative	N
	total	P	N	

True Positive (TP):	Model correctly predicts an assets upward price movement
False Positive (FP):	Model predicts an assets price to go up when it went down
True Negative (TN):	Model correctly predicts an assets downward price movement
False Negative (FN):	Model predicts an assets price to go down when it went up

Derived from TP, FP, TN and FN are *True Positive Rate (TPR)*, *True Negative Rate (TNR)*, *Positive Predictive Value (PPV)*, *Negative Predictive Value (NPV)* and *Accuracy*,

$$\textit{True Positive Rate} = \frac{\textit{True Positive}}{\textit{True Positive} + \textit{False Negative}}$$

$$\textit{True Negative Rate} = \frac{\textit{True Negative}}{\textit{True Negative} + \textit{False Positive}}$$

$$\textit{Positive Predictive Value} = \frac{\textit{True Positive}}{\textit{True Positive} + \textit{False Positive}}$$

$$\textit{Negative Predictive Value} = \frac{\textit{True Positive}}{\textit{True Positive} + \textit{False Positive}}$$

$$\textit{Accuracy} = \frac{\textit{True Positive} + \textit{True Negative}}{\textit{True Positive} + \textit{True Negative} + \textit{False Positive} + \textit{False Negative}}.$$

TPR (TNR) is the rate of correct positives (negatives) the model predicted out of all actual positives (negatives). PPV (NPV) is the rate of correct positives (negatives) the model predicted out of all positive (negative) predictions the model generated. The accuracy is the rate of correct predictions out of all model predictions.

2.2 Time Series Analysis

This section will briefly cover the topics of auto-regressive models and naive predictors in the context of traditional time series analysis.

2.2.1 Auto-regressive models

The Auto-regressive (AR) model is a way of describing certain time-varying processes. In many cases, these are time-varying processes with stochastic properties, such as those found in nature or finance. Concretely, the auto-regressive model takes chosen prior values of the time series, multiplies them with some constant and adds them together with a stochastic term. This sum will be the AR model's prediction of a future value. Thus, the output variable is linearly dependent on both its own previous values and a stochastic term.

More precisely, we define an auto-regressive model of order n, $AR(n)$, as

$$X_t = \sum_{i=1}^{n} \varphi_i X_{t-i} + \varepsilon_t$$

where $\varphi_1, \ldots, \varphi_n$ are the parameters of the model, and ε_t is white noise.

2.2.2 Naive predictors

A rudimentary predictor, with a simple and seemingly naive approach to produce predictions, is often called a naive predictor. In the setting of predicting the next day's opening price for a stock, a naive predictor could, for example, be constructed so that the prediction is the previous day's closing price. Naive predictors are usually created to serve as benchmarks for the more complex models. The idea behind this is that if the more complex model does not outperform the naive model, it is ineffective. Increasing the complexity of a model should, in general, only be done if it improves the performance of the model. If no performance improvements are apparent, then it is usually better to opt for the model with lower complexity.

2.3 Artificial Neural Networks

Artificial neural networks (ANNs), sometimes known as neural networks (NNs), are mathematical structures that draw inspiration from the biological neural networks that make up brains. More precisely, it is an collection of nodes connected by edges. They are often used to model patterns in data, meaning function approximation, or to model probability distributions, but also to solve classification and regression tasks in supervised learning.

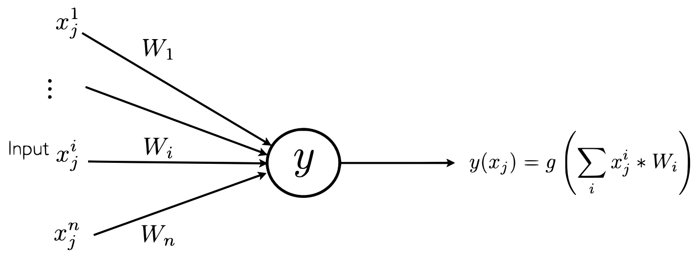

Figure 2.1: Example of a single neuron network (Bringsjord and Govindarajulu, 2021).

Artificial neurons, or nodes, that loosely resemble the neurons in a biological brain, are the foundation of an ANN. Like the synapses in a human brain, each link has the ability to send a signal to neighboring neurons. An artificial neuron can signal neurons that are connected to it after processing signals that are sent to it. The output of each neuron is calculated by some function of the sum of its inputs, and the signal at a connection is a real number. Edges refer to the connections. One commonly used activation function is the Rectified Linear Unit (ReLU), defined as

$$ReLU(x) = max(0, x),\ x \in \mathbb{R}.$$

The weight of neurons and edges often changes as learning progresses. The weight alters a connection's signal intensity by increasing or decreasing it. Neurons may have a threshold, and only send a signal if the combined signal crosses it, much alike how neurons in the human brain functions. This is the case for the ReLU. Neurons are frequently grouped together into layers. Different layers may modify their inputs in different ways. Signals move through the layers, from the first layer, the input layer, to the last layer or neuron, usually called the output layer, a visual representation of this is seen in Figure 2.2 from Graves (2012).

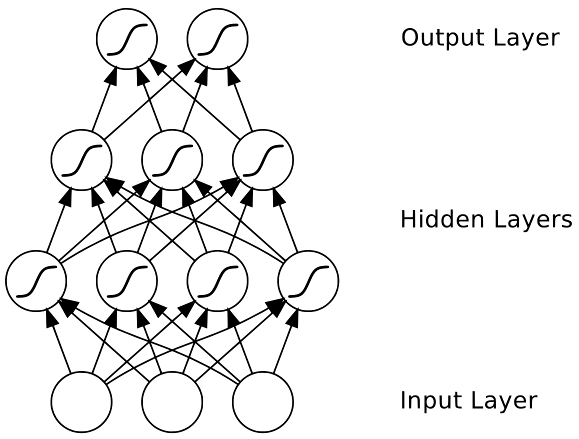

Figure 2.2: A simple neural network with two hidden layers. The S-shaped curves indicate that the activation functions are logistic sigmoids.

Lets now define a basic neural network. The feed forward neural network, with input and output dimensions N and M respectively, is defined by the function

$$\mathbb{R}^N \ni \boldsymbol{x_0} \Rightarrow f(\boldsymbol{x_0}; \boldsymbol{W}) \in \mathbb{R}^M,$$

where $\boldsymbol{x_0}$ is the input and $\boldsymbol{W}$ is the collection of weights parameters. The output of feedforward neural network layer, $\ell \in \{1, \dots, L\}$, is

$$\boldsymbol{h}^{(\ell)}(\boldsymbol{h}^{(\ell-1)}) = \phi^{(\ell)}(\boldsymbol{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}),$$

where $\boldsymbol{h}^{(\ell-1)}$ is the output of the layer preceding $\boldsymbol{h}^{(\ell)}$, $\phi^{(\ell)}$ is the activation function of layer ℓ which operates element wise on the input vector, $\boldsymbol{W}^{(\ell)}$ is the weight matrix, $\boldsymbol{b}^{(\ell)}$ is the biases vector for the layer ℓ, and finally, $f(\boldsymbol{x_0}; \boldsymbol{W}) = \boldsymbol{h}^{(L)}$ where $\boldsymbol{h}^{(0)} = \boldsymbol{x_0}$. The width of the layers in between the input and output layers may have other sizes than N or M.

Another choice of activation function is the Logistic Sigmoid. It is commonly used as the last activation function in the network for binary classification problems. One reason for this is that the output can be interpreted as a probability Goodfellow et al. (2016). The Logistic Sigmoid function is given as,

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

2.3.1 Supervised Learning

Supervised learning (SL) is used to solve issues when the data at hand consists of labelled instances, this means that each input data set has some features and a corresponding label; the case for the problem approached in this thesis. Based on sample input-output pairs, a supervised learning algorithm aims to learn a function that maps the feature vectors (inputs) to the labels (outputs). Each example in supervised learning is a pair that includes an input item, usually transformed into vector, and an intended output value. The function generated by the supervised learning algorithm from the training data can then be used to map new samples. Ideally, this function will be able to accurately determine class labels for these unseen instances (not used for training). To succeed with this, the learning algorithm has to generalize from the training data to hypothetical situations.

Specifically, we have the labeled data pairs, $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_n, y_n)$, where $\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n \in X,\ y_1, y_2 \dots, y_n \in Y$ and the goal is to find the function that connects X and Y.

2.3.2 Recurrent neural networks

In the feed forward neural networks information flows forward, meaning that nodes receive information only from nodes preceding them, and each new input vector x results in one output vector y. If the data is time series data, then this structure will not capture the specific time position, in relation to the whole dataset, that the input vector has. However, in practice it is often desired to capture the time dynamics when dealing with time series data, one way to achieve this is by using the Recurrent Neural Network architecture.

Recurrent Neural Networks differs from feed forward networks in that node input now include node outputs from previous time steps,

$$\boldsymbol{h}_t = \phi_h \left(\boldsymbol{W}_h \boldsymbol{x}_t + \boldsymbol{U}_h \boldsymbol{h}_{t-1} + \boldsymbol{b}_h\right)$$
$$\boldsymbol{y}_t = \phi_y \left(\boldsymbol{W}_y \boldsymbol{h}_t + \boldsymbol{b}_y\right)$$

where t indicates the that it is the node output at time t, $\boldsymbol{U}$ is another parameter matrix, and ϕ_h and ϕ_y are activation functions (Amidi and Amidi).

The basic recurrent neural networks introduced by Elman (1990) in 1990 are capable of learning shorter and simpler patterns, which, unfortunately, is not especially helpful when modelling messy real world data. Methods able to handle longer and more complicated patterns was desired. The reason for this limitation of basic recurrent neural networks is the problem of vanishing or exploding gradients. This is a problem that appear when training deep networks, such as recurrent networks. When training a neural network, each weight is updated proportionally to the partial derivative of the error function with respect to the current weight during each training iteration, see equation 2.1 below. What happens is that this gradient may become increasingly smaller (or bigger) as it passes through the network, ending up so small (or big) that the weights no longer updates (or explode) their value during training. Solving this problem have been attempted numerous times (Graves, 2012), and one successful such attempt is the creation of Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997). The LSTM architecture is one of the most effective approaches for solving the problem along with the Gated Recurrent Unit (GRU) architecture introduced in 2014 by (Cho et al., 2014). GRU is akin to Long Short-Term Memory but has fewer parameters. The LSTM and GRU along with other recurrent neural network architectures are recommended by (Goodfellow et al., 2016) for problems where the inputs and outputs are sequences.

Long-short-term-memory

Figure 2.3 and 2.4 from Graves (2012) displays the LSTM cell and an example of the LSTM architecture respectively. As seen in 2.3, LSTM cells has three gates; the input, the output and the forget gate, these can be seen as continuous analogues of write, read and reset for the cells. These three gates function as nonlinear summing units that assemble activations from both within and outside the block and use multiplication to regulate the activation of the cell (represented by small black circles in Figure 2.3). While the forget gate multiplies the previous state of the cell, the input and output gates multiply the cell's current input and output. Within the cell, no activation is used. Typically, the logistic sigmoid is used as the gate activation function (f), resulting in gate activations between 0 and 1. The input and output activation functions for cells (g and h) are normally logistic sigmoid or tanh functions. Dashed lines in Figure 2.3 represent the weighted connections from the cell to the gates. The block's remaining connections are all unweighted (or equivalently, fixed to 1.0). The output gate multiplication is the only source of outputs from the block to the remainder of the network.

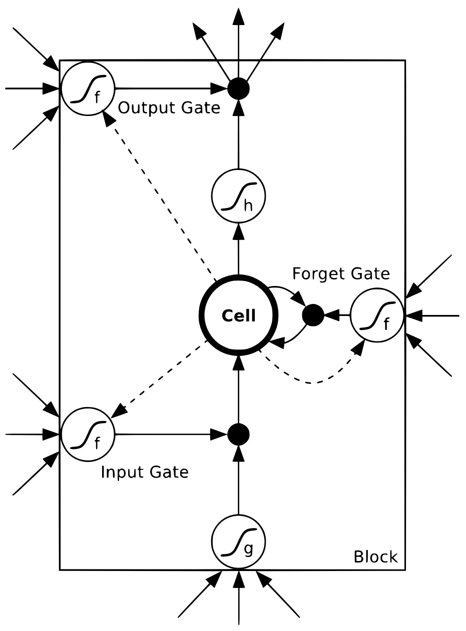

Figure 2.3: A LSTM memory block with one cell.

The network consists of four input units, a hidden layer of two single-cell LSTM memory blocks and five output units. Not all connections are shown. Note that each block has four inputs but only one output (Graves, 2012).

Formally, a LSTM unit is updated as follows. First, every cell state $\boldsymbol{c}_t$ is altered by a forget gate $\boldsymbol{f}_t$, an input gate $\boldsymbol{i}_t$ and an output gate $\boldsymbol{o}_t$. The relation is described by the following set of equations (Li et al., 2018),

$$
\begin{aligned}
\boldsymbol{f}_t &= \phi_g\left(\boldsymbol{W}_f \boldsymbol{x}_t + \boldsymbol{U}_f \boldsymbol{h}_{t-1} + \boldsymbol{b}_f\right)\\
\boldsymbol{i}_t &= \phi_g\left(\boldsymbol{W}_i \boldsymbol{x}_t + \boldsymbol{U}_i \boldsymbol{h}_{t-1} + \boldsymbol{b}_i\right)\\
\boldsymbol{o}_t &= \phi_g\left(\boldsymbol{W}_o \boldsymbol{x}_t + \boldsymbol{U}_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o\right)\\
\tilde{\boldsymbol{c}}_t &= \tanh_c\left(\boldsymbol{W}_c \boldsymbol{x}_t + \boldsymbol{U}_c \boldsymbol{h}_{t-1} + \boldsymbol{b}_c\right)\\
\boldsymbol{c}_t &= \boldsymbol{c}_{t-1} * \boldsymbol{f}_t + \tilde{\boldsymbol{c}}_t * \boldsymbol{i}_t\\
\boldsymbol{h}_t &= \tanh_t\left(\boldsymbol{c}_t\right) * \boldsymbol{o}_t
\end{aligned}
$$

where ϕ is the sigmoid function, $\boldsymbol{W}_*$ and $\boldsymbol{U}_*$ are weight matrices, $\boldsymbol{b}_*$ are bias-vectors, $\boldsymbol{x}_t$ is the input to the cell at time t and, lastly, $\boldsymbol{h_t}$ is the hidden state output and $*$ is element-wise multiplication.

In Figure 2.4 the LSTM memory cells can be seen implemented in a neural network. This is a network with four input units, one hidden layer consisting of two single-cell LSTM memory blocks, and five output units. Note that not every connection is displayed and that each block has four inputs but just one output.

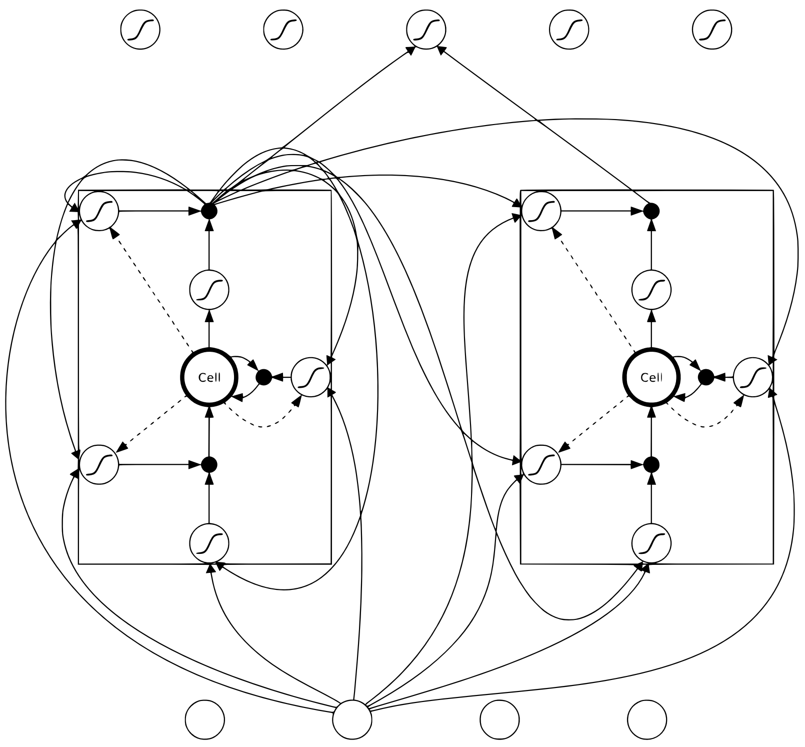

Figure 2.4: A simplified LSTM network consisting of two of the LSTM memory cells seen in Figure 2.3 above (Graves, 2012).

Gated recurrent unit

The Gated Recurrent unit memory cell was developed as an answer to if all pieces of the LSTM architecture was really needed in order to capture long term dependencies. GRU proved that they were not. But, despite having fewer pieces, an improvement in performance over LSTMs has been demonstrated on several tasks and datasets. The main difference between GRU and LSTM cells is that a single gating unit controls both the forgetting factor and the decision to update the state unit (Chung et al., 2014). Resulting in that GRU cells only have two gates instead of three as seen in LSTM cells. Having two gates, a reset and a update gate, instead of three means that there will be fewer parameters in GRU networks. The GRU memory cell is updated as follows (Heck and Salem, 2017),

$$\begin{aligned}
\boldsymbol{u}_t &= \phi_g\left(\boldsymbol{W}_u\boldsymbol{x}_t + \boldsymbol{U}_u\boldsymbol{h}_{t-1} + \boldsymbol{b}_u\right)\\
\boldsymbol{r}_t &= \phi_g\left(\boldsymbol{W}_r\boldsymbol{x}_t + \boldsymbol{U}_r\boldsymbol{h}_{t-1} + \boldsymbol{b}_r\right)\\
\hat{\boldsymbol{h}}_t &= \phi_h\left(\boldsymbol{W}_h\boldsymbol{x}_t + \boldsymbol{U}_h\left(\boldsymbol{r}_t * \boldsymbol{h}_{t-1}\right) + \boldsymbol{b}_h\right)\\
\boldsymbol{h}_t &= \left(\boldsymbol{1} - \boldsymbol{u}_t\right) * \hat{\boldsymbol{h}}_t + \boldsymbol{u}_t * \boldsymbol{h}_{t-1}
\end{aligned}$$

where $\boldsymbol{u}_t$ is the update gate, $\boldsymbol{r}_t$ is the reset gate and, as previously, $\boldsymbol{h_t}$ is the hidden state output.

2.3.3 Feature selection and extraction

Feature selection refers to the process of choosing a subset of relevant features for use in developing the model. In the context of time series prediction using neural networks, feature selection would involve identifying a subset of the time series data;

such as particular time steps or certain variables that are believed to have explanatory power of future values. The core idea underlying the use of feature selection techniques is that the data contains redundant features that can be eliminated with minimal loss of information. In the case of predicting the next value in a a time series, one could for example assume that data from a longer time ago is no longer relevant, or have very little explanatory power, when predicting the next value in the same time series. It might also be that one relevant characteristic is redundant in the presence of another relevant characteristic with which it is highly correlated. For instance, if there is auto-regressive properties in the data, this might lead to that older data points could be deemed redundant since some of the information will be present in more recent data points.

Feature extraction generates new features from functions that have the the original features as input. For financial time series data such functions could for example be moving averages, volatility or some of the popular technical indicators like Moving-Average-Convergence-Divergence (MACD), Bollinger-bands or Relative Strength Index (RSI).

2.3.4 Imbalanced data

In classification tasks with an uneven split in the number of samples in each class, this is often referred to as "unbalanced data". This might create problems when training a machine learning model. The training model will spend the majority of its time on instances of one class and not learn enough from the other class because there are so few samples in comparison to the other class. Say one have a severe skew between two classes, that is, less than 1% in one class and the rest in the other class. Then, in the case of a batch size of 128, many batches won't have any samples of the class with less than 1% representation, making the gradients less informative (Google Developers). Batch size refers to the number of training examples used in one iteration.

A way of handling the issue of imbalanced data by downsampling and upweighting. Downsampling is when one samples from the majority class examples, creating a subset, and trains the model on this subset instead of the whole set. This process will improve the imbalance between the sets. If desired, one could sample so that there is an even split between the two classes.

Upweighting means that one adds an example weight to the subset created by downsampling. This weight is equal to the factor by which the downsampling was performed (Google Developers).

2.3.5 Training neural networks

When processing samples that each have a known "input" and "label", neural networks build probability-weighted associations between the two. These associations are then stored in the network's structure in the tunable parameters of the network, such as the weights. In order to train a neural network from a given example, one compares the processed output of the network—often a prediction—against the desired output. The error is in the discrepancy between the two. The network then modifies its weighted associations using this error value and a learning strategy, in practice this implies minimizing a loss function through the use of some optimization algorithm. The neural network will produce outputs that are increasingly comparable to the goal output as modifications are made over time. These modifications are usually made a number of times before the training is stopped as it fulfills certain conditions. This is called this supervised learning.

Without being designed with task-specific rules, neural networks still "learn", as they are trained, by considering a large number of examples. For instance, in time series prediction, they might study sample vectors each containing data points preceding the target data point. If there are learnable patterns in the data, the neural net will be tuned to identify these patterns. When the neural net is later feed with an unseen vector, it will produce a prediction based on the learned patterns from the training data.

Since training a network on a given dataset, $\mathcal{D}$, is essentially an optimization problem with the goal of minimizing a loss function, the training task can thus be posed as finding the weights that will satisfy:

$$\boldsymbol{W}^* = \arg\min_{\boldsymbol{W}} \mathcal{L}(\boldsymbol{W}; \mathcal{D}),$$

where $\mathcal{L}(\boldsymbol{W}; \mathcal{D})$ is the loss function tailored to the task at hand.

An optimization algorithm describes the procedure of identifying the input parameters or arguments for a function so that the minimum or maximum output of the function is found, here that would be the weights, $\boldsymbol{W}$, which minimizes $\mathcal{L}(\boldsymbol{W}; \mathcal{D})$. One such optimization algorithm is Gradient Descent (GD),

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \eta \nabla_{\boldsymbol{W}_t} \mathcal{L}\left(\boldsymbol{W}_t; \mathcal{D}\right), \tag{2.1}$$

where η is the learning rate.

Gradient descent runs through all samples in the training set to do a single parameter update in a particular iteration. This is computationally inefficient and can take a long time when the training data set is large. Therefore, other methods have been designed. With the data often batched, stochastic gradient descent (SGD) or mini-batch stochastic gradient descent can be used. Using either one or a subset of training samples to update a parameter in a particular iteration, respectively.

Backpropagation through time

To compute the gradients in order to tune the parameters of a recurrent neural network, backpropagation through time (BPTT) (Werbos, 1990) is often applied. Lets walk through it briefly using the same notation as previously, for the interested a gentle introduction is found in Chen (2016).

For clarity, a loss function $\mathcal{L}(\boldsymbol{W}; \mathcal{D})$ is set, let it be the cross-entropy defined as

$$\mathcal{L} = -\sum_t y_t \cdot \log \hat{y}_t.$$

where y_t is the label to predict and $\hat{y}_t$ is the prediction obtained from the model.

Recall that the goal of this procedure is to find the parameters that minimise $\mathcal{L}$. To accomplish this the derivative $\frac{\partial \mathcal{L}}{\partial m_t}$ must be obtained, where m_t is $b_y + W_y h_t$. From (Chen, 2016) we have,

$$\frac{\partial \mathcal{L}}{\partial m_t} = \hat{y}_t - y_t$$

Now, the derivatives with respect to U_h can be found since a RNN uses the same U_h in each time step. Consider just the derivative of the loss function at time step $t+1$, that is

$$\mathcal{L}_{t+1} = -y_{t+1} \log \hat{y}_{t+1}$$

then

$$\begin{aligned}\frac{\partial \mathcal{L}_{t+1}}{\partial U_h} &= \frac{\partial \mathcal{L}_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial W} \\ &= \sum_{k=1}^{t} \frac{\partial \mathcal{L}_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_k} \frac{\partial h_k}{\partial W},\end{aligned}$$

where BPTT was used in the second equality. The derivative with respect to U_h is arrived at by summing up at all time steps,

$$\frac{\partial \mathcal{L}}{\partial U_h} = \sum_{t=1}^{N-1} \sum_{k=1}^{t+1} \frac{\partial \mathcal{L}_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_k} \frac{\partial h_k}{\partial U_h}$$

assuming $t = 1, \ldots, N$.

Now, lets find the derivative with respect to W_h. Start with the derivative of the last time step,

$$\frac{\partial \mathcal{L}_{t+1}}{\partial W_h} = \frac{\partial \mathcal{L}_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial W_h} = \sum_{k=1}^{t+1} \frac{\partial \mathcal{L}_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_k} \frac{\partial h_k}{\partial W_h}.$$

and then sum up the derivatives at all previous time steps,

$$\frac{\partial \mathcal{L}}{\partial W_h} = \sum_{t=1}^{N-1} \sum_{k=1}^{t+1} \frac{\partial \mathcal{L}_{t+1}}{\partial z_{t+1}} \frac{\partial z_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_k} \frac{\partial h_k}{\partial W_h}.$$

Finally, we calculate the derivatives with respect to b_y and W_y as follows,

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b_y} &= \sum_{t=1}^{N-1} \frac{\partial \mathcal{L}}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial b_y} \\ \frac{\partial \mathcal{L}}{\partial W_y} &= \sum_{t=1}^{N-1} \frac{\partial \mathcal{L}}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial W_y}\end{aligned}$$

Regularization

A common issue appearing when training neural networks is overfitting, it is a serious problem and can be challenging to prevent. For example, ensemble regularization methods, which combine the predictions of numerous neural nets at test time, might take a long time for large nets. Dropout is a regularization method for dealing with this issue. The main concept is to randomly remove nodes and their connections, with some probability hyperparameter p, from the neural network during training. This has been proved to improve the performance of neural networks while reducing the computational effort needed to do so (Srivastava et al., 2014). Figure 2.5 from Srivastava et al. (2014) visualizes the method.

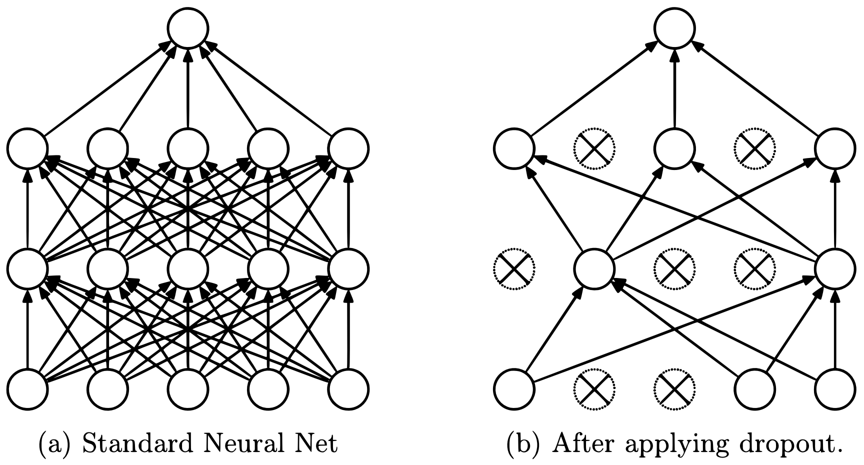

Figure 2.5: Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right: An example of a net with dropout. Crossed units have been dropped (Srivastava et al., 2014).

Other ways to achieve regularization include choosing smaller batch sizes and in some cases batch normalization, see section below.

Batch normalization

Batch normalization is a reparameterization of the model in a way that introduces an addition and multiplication to the hidden units during training. The primary purpose of batch normalization is to improve optimization, but to avoid encountering an undefined gradient in the calculations, noise is added, and this noise can also have a regularizing effect; sometimes even making dropout unnecessary (Goodfellow et al., 2016).

Batch normalization is performed by applying the following procedure (Ioffe and

Szegedy, 2015),

```
Input:   Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$
         Parameters to be learned: $\gamma, \beta$
Output:  $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                                    // mini-batch mean
$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2$            // mini-batch variance
$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$                      // normalize
$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$                // scale and shift
```

where ϵ is a constant added for numerical stability. The last step is done in order to maintain expressive power of the network. Instead of just replacing the hidden unit activations x_i with the normalized $\widehat{x}_i$, they are instead replaced by y_i. Now, since γ and β are learnable parameters, this allows y_i to have any mean and standard deviation (Goodfellow et al., 2016).

Binary cross-entropy

When dealing with probability values, $p \in [0, 1]$, common loss functions such as the squared residual with $loss = (1 - p)^2$ are inefficient due to the small changes in loss when the probability p changes. Since the step size in backpropagation depends in part on the derivative of the loss function, a monotonically decreasing loss function with large loss for bad predictions and small loss for good predictions is desired. The derivative of the cross-entropy loss function will be relatively large compared to that of the squared residual loss function for bad predictions. Hence, the cross-entropy loss function will improve training as compared to squared residual since the step sizes during backpropagation will be larger. Cross-entropy can be written as,

$$\textit{Cross-entropy} = -\frac{1}{N}\sum_{i=1}^{N} \left(p_i \log(\hat{p}_i) + (1 - p_i) * \log(1 - \hat{p}_i)\right),$$

and for $N = 1$ one finds binary cross-entropy (Nielsen, 2015).

The Adam optimizer

The Adam optimization algorithm is an extension to stochastic gradient descent. There are a large number of different optimizers one could use, and which one is best will depend on the problem at hand. However, some optimizers might be superior in some cases as compared to others. The Adam optimizer is an algorithm that performs well in a number of different settings. According to Ruder (2016) the Adam optimizer might be the best overall choice under certain conditions. The Adam optimizer is also recommended as the default algorithm to use in the course CS231n at Stanford (CS231n).

One way to grasp the Adam algorithm is to step through the algorithm of it. The algorithm can be written as (Kingma and Ba, 2014),

Algorithm 1 The Adam algorithm

```
    Input: $f(W)$, $W_0$, $\eta$, $\beta_1$, $\beta_2$, $\epsilon$
    Output: $W_t$
 1: procedure
 2:     $m_0 \leftarrow 0$
 3:     $v_0 \leftarrow 0$
 4:     $t \leftarrow 0$
 5:     while $W_t$ not converged do
 6:         $t \leftarrow t + 1$
 7:         $g_t \leftarrow \nabla_{W_t} f_t (W_{t-1})$
 8:         $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
 9:         $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
10:         $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
11:         $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
12:         $W_t \leftarrow W_{t-1} - \eta \cdot \widehat{m}_t / \left(\sqrt{\widehat{v}_t} + \epsilon\right)$
        return $W_t$
```

where, $f(W)$ is the objective function with weights (parameters) W, W_0 are the initial weights, η is the learning rate and β_1, β_2 and ϵ are hyperparameters.

What happens is that the first and second moment vectors along with the timestep are initialized. Then the while loop is entered, here, t is updated and the gradient with respect to the objective function at time t is found. Then the biased first and second moments are updated. After that the bias-corrected first and second moment estimates are computed. Finally, the weights are updated. This is done until the weight matrix has converged. The resulting weights are then returned.

In essense, the Adam algorithm computes and keeps track of the biased first and second moment of the gradients, or mean and variance respectively, as follows

$$\begin{aligned} m_{t+1} &= \beta_1 m_t + (1 - \beta_1) \nabla_{\boldsymbol{W}_t} \mathcal{L} (\boldsymbol{W}_t; \mathcal{D}) \\ v_{t+1} &= \beta_2 v_t + (1 - \beta_2) \left(\nabla_{\boldsymbol{W}_t} \mathcal{L} (\boldsymbol{W}_t; \mathcal{D})\right)^2 , \end{aligned}$$

where β_1 and β_2 are usually set to 0.9 and 0.999 respectively (Kingma and Ba, 2014). The moments are initialised to zero as seen in the code above. The unbiased moment estimates are then calculated,

$$\begin{aligned} \hat{m}_{t+1} &= \frac{m_{t+1}}{1 - \beta_1^t} \\ \hat{v}_{t+1} &= \frac{v_{t+1}}{1 - \beta_2^t} . \end{aligned}$$

Finally, the weights are updated,

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \eta \frac{1}{\sqrt{\hat{v}_{t+1}} + \epsilon} \hat{m}_{t+1},$$

where ϵ is needed to numerically stabilize the calculations, it is usually set to 10^{-8}.

Walk-forward-validation

To find the best parameters of a model predicting financial time series, walk-forward-validation is often used. A time frame in a data series is used to optimize the model. The remaining data is set aside for out-of-sample testing. The period covered by the out of sample test is added to the in-sample time window, and the procedure is repeated. Finally, the trading strategy is evaluated on the last pre-determined out-of-sample testing set that has been reserved for a final evaluation of the model. The procedure is performed as follows; first, split the full dataset into two sets, one in-sample set and one out-of-sample set. Then, in the case of a simple time series, starting at the start x_0 of the in-sample set, the model is trained on the number n previous sample values used as input to predict the next value. The model is then used to make a prediction p_n for the next time step. This prediction is evaluated against the known value (label) l_n. One step forward is then taken and the process is repeated; hence the name walk-forward-validation. The process can be understood as moving along the time series one-time step at a time.

3

Methodology

This section explains...

3.1 Software

Microsoft Excel was utilized to extract and handle the data. The analysis was conducted in Python3 using the Jupyter Notebook platform within the Visual Studio Coding code editor. Several packages were imported. The most common packages were Pandas, NumPy, PyPlot, and Matplotlib.

3.2 Data

Historical time series data for closing prices on five assets were received from Handelsbanken Fonder. Information about the data can be found in table 3.1.

Table 3.1: Data table

Series Name	Frequency	Unit	Start Date	End Date
RX1	Daily	Price	2000-01-04	2022-10-25
TY1	Daily	Price	2002-01-04	2022-10-25
IK1	Daily	Price	2009-09-15	2022-10-25
OE1	Daily	Price	2002-01-04	2022-10-25
DU1	Daily	Price	2002-01-04	2022-10-25

3.2.1 Data storage and extraction

The data was collected from the Bloomberg Terminal and Handelsbankens internal systems. The data was initially stored in Excel files. These files were instead saved as csv files to reduce file size and improve load time into Python. The csv files were then loaded into a Pandas DataFrame. This DataFrame was then manipulated to only contain the data of interest. The same overall process applies to all model data used in this thesis.

3.2.2 Features

The features used were previous closing prices from the time series itself. The model was extended to handle inputs from multiple other time series. When incorporating other time series as inputs, previous closing prices from these time series were used.

3.2.3 Train-validation-test split

The data was split up into three sets: training, validation and testing. The training set was used to train the models. The validation set was then used evaluate the model's accuracy and to tune the hyper-parameters. The results from the validation accuracy were used as a base to evaluate which model performed the best. The improved models were then retrained on the training data and finally evaluated on the test data.

3.2.4 Pre-processing

In order to use the data in practice, a number of data manipulation steps are needed. A underlying premise of neural network learning is that training and test data come from the same probability distribution. That is, x_{train} and x_{test} should be outcomes of X if X is a random variable. This not always the case for time series data since some processes might have non-stationary properties. Asset prices are among the examples of processes that usually have non-stationary properties. Stock prices have as an example typically an upward drift of both mean and variance. Monthly annualized volatilities are usually lower than daily annualized volatilities; while annual volatilities are higher. As a result, prices typically exhibit short-term reversion and long-term trending. Additionally, volatility normally decreases gradually before abruptly increasing. Therefore, to arrive at something more stationary, the asset price data was transformed into returns,

$$r_t = (p_t - p_{t-1})/p_{t-1},$$

and scaled to be in the interval $x \in [0, 1]$ using preprocessing.scale() from the scikit-learn library. The scaling was done separately for the training, validation and test set to not introduce data leakage. After this, the data was split into as many input and target series as possible for each time series set. The input series consists of the n previous data points of the target located at position $n + 1$ in the original time series. There is one input series for each target, and the number of targets in a time series is $t = length(series) - n$. Resulting from this will be t sets each consisting of one input series and one target. These sets will now be split into two groups, one with all the negative targets, "downs", and one with all the positive, "ups". If the target is negative it is replaced with an 0, and the remaining, positive targets, are replaced with ones. Instead of having specific values as targets, the described process have reduced the targets to a binary choice of 1 or 0, representing up and down respectively. The number of elements in each group, "ups" or "downs", is counted and the group with the most elements is reduced so that it has the same size as the smaller group. Finally, the two groups are merged and shuffled. Note that "targets" here are interchangeable with "labels".

3.3 Neural Network Model Structure

The neural network model structures used were Long Short-Term Memory and Gated Recurrent Units. Both models consisted of 4 layers with 128 LSTM or GRU nodes respectively followed by one dense layer of 32 ReLU nodes and finally a single sigmoid output neuron. Added between each layer was dropout with $p = 0.25$ and batch normalization. Hyperparameters were *learning-rate* = 1e−4, *decay* = 1e−6, *epochs* = 30 and *batch-size* = 8. A binary cross-entropy loss function was used for training. The model for each asset was trained on the training data derived from that same asset, that is, the neural model used to trade for example RX1 was trained on the training data generated from RX1. The 60 previous returns was used to predict the next return, thus to predict r_t, $(r_{t-1}, r_{t-2}, ..., r_{t-60})$ was used as input to the model. Dropout (Geoffrey E. Hinton, US Patent 9,406,017 B2, August 2, 2016) was implemented in accordance with the recommendations to select the least flexible model that produces comparable cross-validation results (Zhang et al., 2020). Zhang et al. (2020) also notes that using dropout improves the results for financial applications, especially in the case of complex network architectures.

3.3.1 Hyper-parameter optimization

Since K-fold cross-validation fails in finance (De Prado, 2018), walk-forward-validation is used for hyper-parameter tuning instead. Therefore, any hyper-parameter optimization is done using the validation set; leaving the test set for a final performance evaluation and reducing data leakage. The test set is allocated the last 10% of the full dataset with the remaining 90% being split between training and validation as 90% and 10% respectively.

3.4 Trading Algorithms

3.4.1 Time Series Analysis

In order to benchmark the more advanced models, very simple models were implemented and evaluated. The performance of these models where then compared to the performance of the advanced models. Specifically, one auto-regressive model of order one, AR(1), and one of order 10, AR(10), was used; both of these are to be viewed as naive predictors.

The AR(1) model simply takes the most recent return as its prediction, that is $y_t = y_{t-1}$, which implies that the direction (up or down) of the last return will be the predicted direction of the next return. The AR(10) models prediction is the weighted average of the last $n = 10$ returns,

$$y_t = \frac{y_{t-1} + y_{t-2} + \cdots + y_{t-n}}{n}.$$

again this means that if this weighted average is bigger than zero the model will predict that the next return is positive and vice versa.

3.5 Benchmarks

3.5.1 Economical benchmarks

Benchmarking can be done after the models have been used to predict asset direction and the trading algorithm has converted these predictions into trading performance. The resulting trading performance is used to find the Sharpe-ratio, Sortino ratio, maximal drawdown and gross return. On the request of Handelsbanken, the risk free rate (rf) was set to $rf = 0.0$.

Since (Lo, 2002) found that the traditional method of calculating Sharpe-ratios results in inaccurate readings, another method will be used in this thesis. Instead, the annual return and volatility will be calculated from the portfolio development 4.2 directly. The following code displays how the Sharpe-ratio was calculated,

```python
def sharpe_ratio(portfolio_values, risk_free_rate):
    tot_return = gross_return(portfolio_values) - risk_free_rate
    years_held_invest = len(portfolio_values) / 252
    tot_return_annual = pow(1 + tot_return, 1/years_held_invest) - 1
    volatility_yearly = yearly_standard_deviation(portfolio_values)
    return tot_return_annual / volatility_yearly

def portfolio_yearly_standard_deviation(portfolio_values):
    every_fifth_value = portfolio_values[0::5]
    weekly_returns = returns(every_fifth_value)
    yearly_variance = weekly_returns.var() * 52
    yearly_standard_deviation = np.sqrt(yearly_variance)
    return yearly_standard_deviation
```

3.5.2 Statistical benchmarks

Statistical benchmarking (classification) is performed using the model outputs directly, hence the labeling emphasising a difference from the financial benchmarks. The financial benchmarks are affected by the actual returns. If the model is wrong for a prediction where there is huge move, this will significantly affect the financial performance negatively. Due to the cumulative nature of returns this means that just a few decisions can permeate the models financial performance. This results in making financial performance potentially misleading. Therefore, it seems meaningful to also observe benchmarks not affected by this randomness. Calculated for this task are true negatives (TN), true positives (TP), false negatives (FN) and false positives (FP). Derived from these are then true positive rate (TPR), true negative rate (TNR), positive predictive value (PPR) and negative predictive value (NPR); all of which are independent from the returns.

4

Results

In the following chapter...

4.1 Benchmarks

Averaged benchmark results. The Trend model performs the best across almost (excluding TPR, NPR, PPV and NPV) all measures; it has the highest accuracy, Sharpe ratio, Sortino ratio and gross return and the lowest maximal drawdown. Second best is the GRU model when observing the same measures excluding maximal drawdown where the naive model is slightly better.

Table 4.1: Averaged benchmarking results.

Model	Sharpe Ratio	Sortino Ratio	Gross Return (%)	Maximal Drawdown (%)	True Positive Rate (%)	True Negative Rate (%)	Positive Predictive Value (%)	Negative Predictive Value (%)	Accuracy (%)
Lstm	0.37	0.53	4.43	(9.11)	50.3	50.4	47.0	53.6	50.3
Gru	0.87	1.47	10.77	(5.90)	48.5	55.0	48.6	54.9	52.0
Trend	1.28	2.21	15.04	(4.72)	38.4	64.7	49.0	54.4	52.4
Naive	0.55	0.90	4.43	(5.80)	48.2	54.8	48.3	54.7	51.7
Hold	(1.02)	(2.05)	(14.9)	(15.8)	-	-	-	-	-

4.2 Trading Performance

In Figure 4.1 averaged trading performance is plotted, LSTM models are found in red. The Trend model in green has the highest gross return over the period. The simple naive model in purple has the second highest gross return, outperforming the neural model.

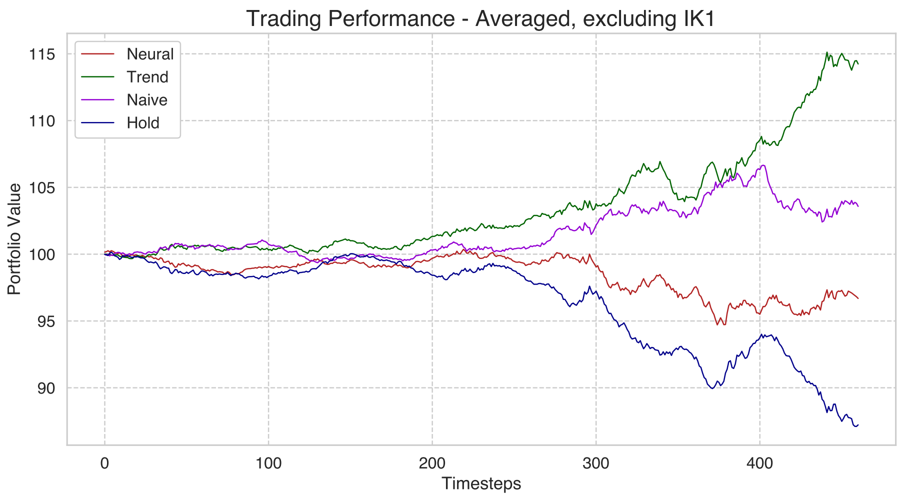

Figure 4.1: Averaged trading performance for the LSTM models plotted in red, excluding IK1.

IK1 LSTM trading performance and neural network model output is seen below, akin results for the other assets (RX1, TY1, OE1 and DU1) are found in Appendix A.1.1.

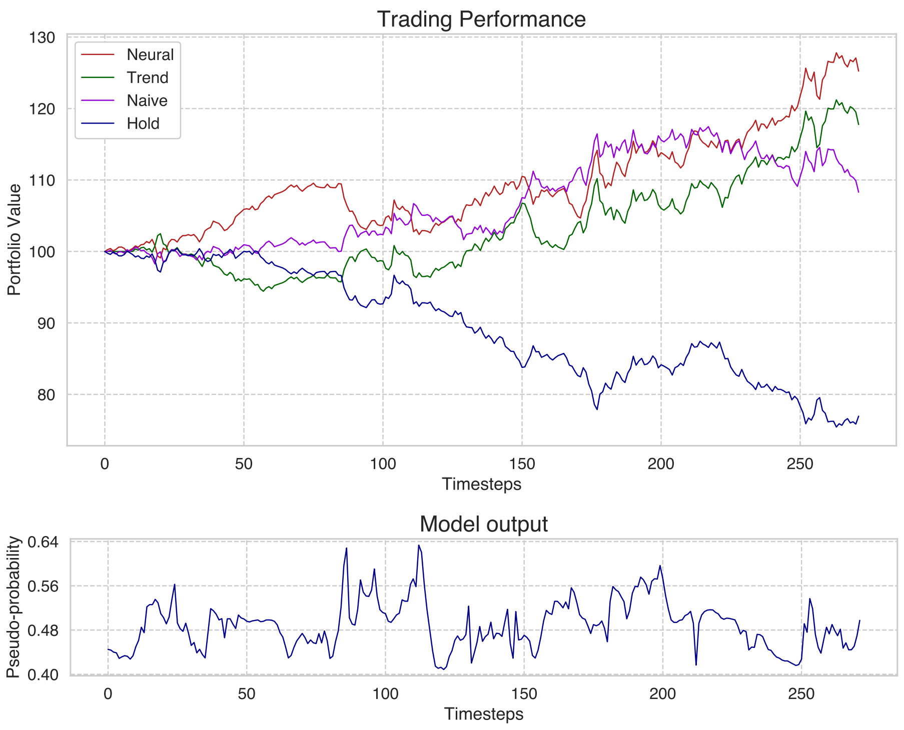

Figure 4.2: Trading performance on the IK1 data set along with the neural network model's output for the LSTM model. Note that the number of time steps (trading days) are fewer in this data set.

In Figure 4.3 averaged trading performance is plotted, GRU models are seen in red. The Trend model in green has the highest gross return over the period followed by the Neural model. Note that the Naive and Trend models performance are identical to to those in Figure 4.1, this is expected since these models are deterministic.

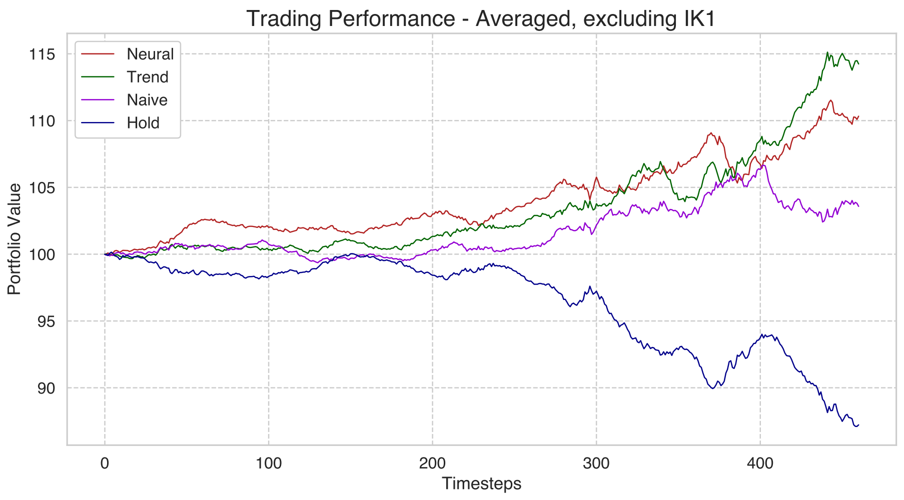

Figure 4.3: Averaged trading performance for the GRU models plotted in red, excluding IK1.

IK1 GRU trading performance and the neural network model's output is seen below, akin results for the other assets (RX1, TY1, OE1 and DU1) are found in Appendix A.1.2.

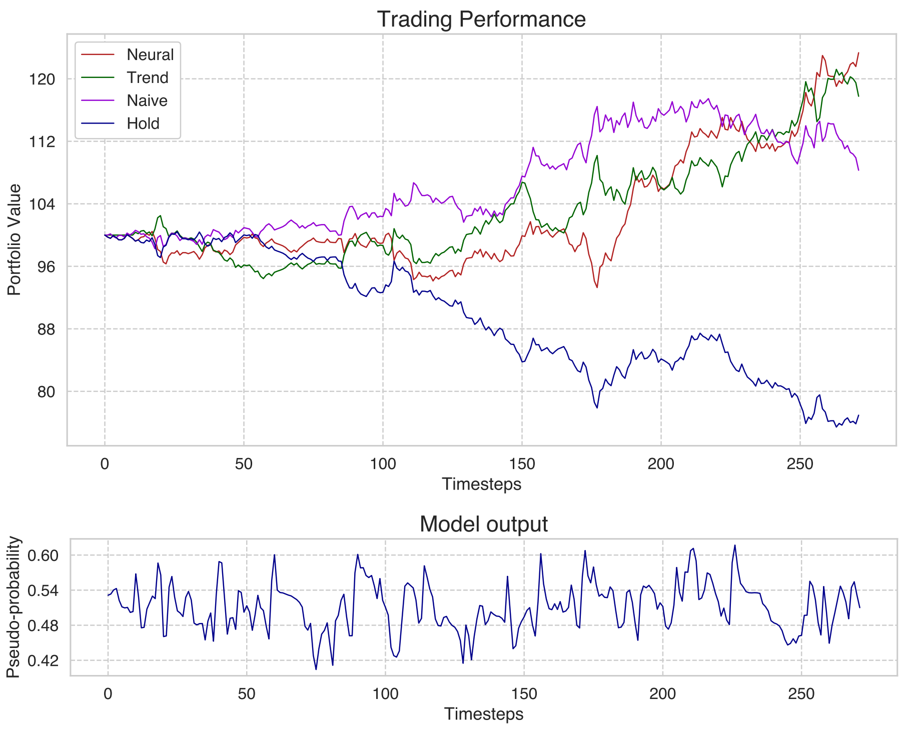

Figure 4.4: Trading performance on the IK1 data set along with the neural network model's output for the GRU model.

Table 4.2: Number of trades and percentage of days long or short.

Data	Model	Number of trades	Days Long (%)	Days Short (%)
RX1	Lstm	79	35.7	64.3
	Gru	70	27.1	72.9
	Trend	58	37.8	62.2
	Naive	230	44.5	55.5
TY1	Lstm	65	53.4	46.6
	Gru	104	42.7	57.3
	Trend	62	35.3	64.7
	Naive	223	47.6	52.4
IK1	Lstm	36	37.1	62.9
	Gru	54	61.0	39.0
	Trend	45	30.5	69.5
	Naive	131	41.7	58.3
OE1	Lstm	84	47.2	52.8
	Gru	92	38.5	61.5
	Trend	54	40.0	60.0
	Naive	228	45.0	55.0
DU1	Lstm	49	70.9	29.1
	Gru	74	69.9	30.1
	Trend	60	37.6	62.4
	Naive	216	52.2	47.8
Average across all data	Lstm	62.6	49.9	50.1
	Gru	78.8	46.7	53.3
	Trend	55.0	36.8	63.2
	Naive	207	46.6	53.4

Table 4.2 displays the number of trades taken by each model and the percentage of days that the model held a long or a short position. When observing the averaged results it can be seen that the Trend model took the least amount of trades followed by the Lstm model. The number of long and short days are almost the same for the Lstm, Gru and Naive models; in contrast to the Trend models which on average held a short position 63% of the days.

5

Conclusion & Discussion

5.1 Conclusion

Based on the findings of this thesis, no definitive conclusions can be drawn regarding the efficacy of artificial neural networks in reliably forecasting expected returns of credit futures and utilizing them in trading algorithms. While the ANN models proposed in this study did not outperform simple predictors, it is important to note that generalization beyond the specific models and time series examined here is challenging. This is primarily due to the multitude of design options available when building neural networks, including input data selection, neural network type, architecture, regularization approach, and parameter selection, among other factors. It is conceivable that with the appropriate design, an ANN model could improve return estimations and produce a trading algorithm that outperforms benchmark models. Further research is needed to explore the potential of ANNs in this domain.

5.2 Discussion

The trend model outperformed the other models tested in this study. However, its simplicity makes it enticing to question whether its success was simply due to luck. It is worth noting, though, that other simple prediction models have also yielded impressive results in different markets. For instance, Bill Benter reportedly made close to a billion dollars by using linear regression models to place bets on horses in Hong Kong (Chellel, 2018). Moreover, there are multiple studies indicating that trend models can outperform markets as well (Johnson, 2002), (Fama and French, 2012), (Barroso and Santa-Clara, 2015), (Chan et al., 1996).

However, it is possible that the superior performance of the trend model was due to it holding a short position for 63% of the days during a period when the underlying asset was trending downward. Additionally, the model's accuracy was just 52.4%, which suggests that its impressive results were only specific to that period. As such, it is difficult to draw any firm conclusions about whether the model could consistently produce such performance in other market climates based solely on the results of this study.
The validation loss that remained constant throughout the training process, as depicted in A.3, may be a result of trying to fit on a signal that appears as noise due to the high level of complexity in the data. In such cases, any predictions made

by the model, no matter how plausible they may seem, could be spurious. This is supported by the fact that both the naive and trend models performed as well as, or even better than, the neural models. However, it is worth noting that the downward trend of the underlying asset during the test period may have benefited the trend and naive models. Therefore, to better understand the effectiveness of the models, evaluation on test periods with both sideways and upward trends is necessary. In summary, although the results of this study do not provide conclusive evidence for the efficacy of neural models in predicting the returns of credit futures, the findings highlight the complexity of the problem and the importance of careful evaluation when using neural networks.

5.3 Future work

Multiple questions and ideas emerged throughout the thesis. In the hopes of inspiring future thesis writers I'll present some of these here,

- How do the algorithms perform if thresholds are set where trades are only taken if the "probability" is above or under some value, else it does nothing?
- If predictions the day before larger movements are evaluated, are these predictions in any way indicating that something bigger might happen the next day as compared to the other predictions?
- If the models are trained on multiple different assets, will they be able to predict one of them? Might the models' performance to predict one of them improve as compared to only training on the asset itself?
- Could multiple assets as inputs for the prediction of one of them improve results?
- How do the models perform for shorter time frame data such between 1-15m ticks? Here, one might also weight more recent data higher than older data in order to continually capture the most recent market dynamics.
- Is there a way to quantify market dynamics and train another model to identify which dynamics prevail given some input data? And can outputs from this side-model then be used as inputs to the trading algorithm?
- It might be an idea to implement information about the performance from a longer time ago, such as 1-5 years, possibly lesser as one goes further back but thematically accurate so that the model have some sense of if the recent few years have been positive or negative.
- The model could be continuously trained with new data to tune it as new market data become available with time.
- Weigh the allocation size depending on how certain the model is on a trade as determined by how much the output deviates from 0,5.
- Leverage the domain-specific knowledge of experts to incorporate additional relevant input datasets.

Bibliography

A. Amidi and S. Amidi. Cs230 - deep learning. URL https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks#.

P. Barroso and P. Santa-Clara. Momentum has its moments. *Journal of Financial Economics*, 116(1):111–120, 2015.

L. Blume, D. Easley, and M. O'hara. Market statistics and technical analysis: The role of volume. *The journal of finance*, 49(1):153–181, 1994.

S. Bringsjord and N. S. Govindarajulu. Artificial Intelligence. In E. N. Zalta and U. Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2021. URL https://plato.stanford.edu/Archives/win2021/entries/artificial-intelligence/neural-nets.html.

L. K. Chan, N. Jegadeesh, and J. Lakonishok. Momentum strategies. *The Journal of Finance*, 51(5):1681–1713, 1996.

P. Chatigny, R. Goyenko, and C. Zhang. Asset pricing with attention guided deep learning. *Available at SSRN 3971876*, 2021.

K. Chellel. The gambler who cracked the horse-racing code. *Bloomberg L.P.*, 2018.

A.-S. Chen, M. T. Leung, and H. Daouk. Application of neural networks to an emerging financial market: forecasting and trading the taiwan stock index. *Computers & Operations Research*, 30(6):901–923, 2003.

G. Chen. A gentle tutorial of recurrent neural network with error backpropagation. *arXiv preprint arXiv:1610.02583*, 2016.

K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

S. U. CS231n. Cs231n convolutional neural networks for visual recognition. URL https://cs231n.github.io/neural-networks-3/.

M. L. De Prado. *Advances in financial machine learning*. John Wiley & Sons, 2018.

D. N. Dreman and M. A. Berry. Overreaction, underreaction, and the low-p/e effect. *Financial Analysts Journal*, 51(4):21–30, 1995.

J. L. Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

D. Enke and S. Thawornwong. The use of data mining and neural networks for forecasting stock market returns. *Expert Systems with applications*, 29(4):927–940, 2005.

E. F. Fama. The behavior of stock-market prices. *The journal of Business*, 38(1): 34–105, 1965.

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

E. F. Fama and K. R. French. Size, value, and momentum in international stock returns. *Journal of financial economics*, 105(3):457–472, 2012.

K. Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130, 1988.

I. S. N. S. Geoffrey E. Hinton, Alexander Krizhevsky. System and method for addressing overfitting in a neural network, US Patent 9,406,017 B2, August 2, 2016.

S. Ghoshal and S. Roberts. Thresholded convnet ensembles: neural networks for technical forecasting. *Neural Computing and Applications*, 32:15249–15262, 2020.

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

Google Developers. Imbalanced data. URL https://developers.google.com/machine-learning/data-prep/construct/sampling-splitting/imbalanced-data.

A. Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer, 2012.

S. J. Grossman and J. E. Stiglitz. On the impossibility of informationally efficient markets. *The American economic review*, 70(3):393–408, 1980.

J. C. Heck and F. M. Salem. Simplified minimal gated unit variations for recurrent neural networks. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1593–1596. IEEE, 2017.

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

J. Huang, J. Chai, and S. Cho. Deep learning in finance and banking: A literature review and classification. *Frontiers of Business Research in China*, 14(1):1–24, 2020.

J. M. Hutchinson, A. W. Lo, and T. Poggio. A nonparametric approach to pricing and hedging derivative securities via learning networks. *The journal of Finance*, 49(3):851–889, 1994.

D. Hübner. Ep perspective on today's regional policy and the relevance of financial engineering instruments, 29-30 November 2010.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

R. A. Ippolito. Efficiency with costly information: A study of mutual fund performance, 1965–1984. *The Quarterly Journal of Economics*, 104(1):1–23, 1989.

T. C. Johnson. Rational momentum effects. *The Journal of Finance*, 57(2):585–608, 2002.

J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

S. H.-B. Lauren Tara LaCapra. Goldman to close global alpha fund after losses. 2011. URL https://www.reuters.com/article/us-goldmansachs-hedgefund-idUSTRE78F28Y20110916.

H. Li, Y. Shen, and Y. Zhu. Stock price prediction using attention-based multi-input lstm. In *Asian conference on machine learning*, pages 454–469. PMLR, 2018.

I. E. Livieris, E. Pintelas, and P. Pintelas. A cnn–lstm model for gold price time-series forecasting. *Neural computing and applications*, 32(23):17351–17360, 2020.

A. W. Lo. The statistics of sharpe ratios. *Financial analysts journal*, 58(4):36–52, 2002.

A. W. Lo, H. Mamaysky, and J. Wang. Foundations of technical analysis: Computational algorithms, statistical inference, and empirical implementation. *The journal of finance*, 55(4):1705–1765, 2000.

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

M. A. Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

L. Pospisil and J. Vecer. Pde methods for the maximum drawdown. *Journal of Computational Finance*, 12(2):59–76, 2008.

B. B. B. M. Raffaele Savi, Jeff Shen. The evolution of active investing finding big alpha in big data. *Blackrock*, 2015.

S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016. URL http://arxiv.org/abs/1609.04747.

U. G. Securities and E. Commission. Smart beta, quant funds and other non-traditional index funds. URL https://www.investor.gov/introduction-investing/investing-basics/investment-products/mutual-funds-and-exchange-traded-3.

W. F. Sharpe. Mutual fund performance. *The Journal of business*, 39(1):119–138, 1966.

F. A. Sortino and R. Van Der Meer. Downside risk. *Journal of portfolio Management*, 17(4):27, 1991.

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

A. Timmermann and C. W. Granger. Efficient market hypothesis and forecasting. *International Journal of forecasting*, 20(1):15–27, 2004.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

B. Widrow, D. E. Rumelhart, and M. A. Lehr. Neural networks: applications in industry, business and science. *Communications of the ACM*, 37(3):93–106, 1994.

G. Zhang, B. E. Patuwo, and M. Y. Hu. Forecasting with artificial neural networks:: The state of the art. *International journal of forecasting*, 14(1):35–62, 1998.

Z. Zhang, S. Zohren, and S. Roberts. Deeplob: Deep convolutional neural networks for limit order books. *IEEE Transactions on Signal Processing*, 67(11):3001–3012, 2019.

Z. Zhang, S. Zohren, and S. Roberts. Deep reinforcement learning for trading. *The Journal of Financial Data Science*, 2(2):25–40, 2020.

G. Zuckerman. The making of the world's greatest investor. 2019a.

G. Zuckerman. *The man who solved the market: How Jim Simons launched the quant revolution.* Penguin, 2019b.

Appendix A

A.1 Portfolio Development For Each Asset

The figures below display the portfolio value for each model for the different assets. The performance of the neural network model (Neural) is plotted in red, this performance is seen along with the benchmark models (Trend and Naive) and the underlying asset itself (Hold). Just below the trading performance is the neural network model's output. The neural model's accuracy and loss on the training and validation set can be found in section A.3.

A.1.1 LSTM models trading performance

First, the LSTM models performance. The GRU models performance are found in the next subsection A.1.2.

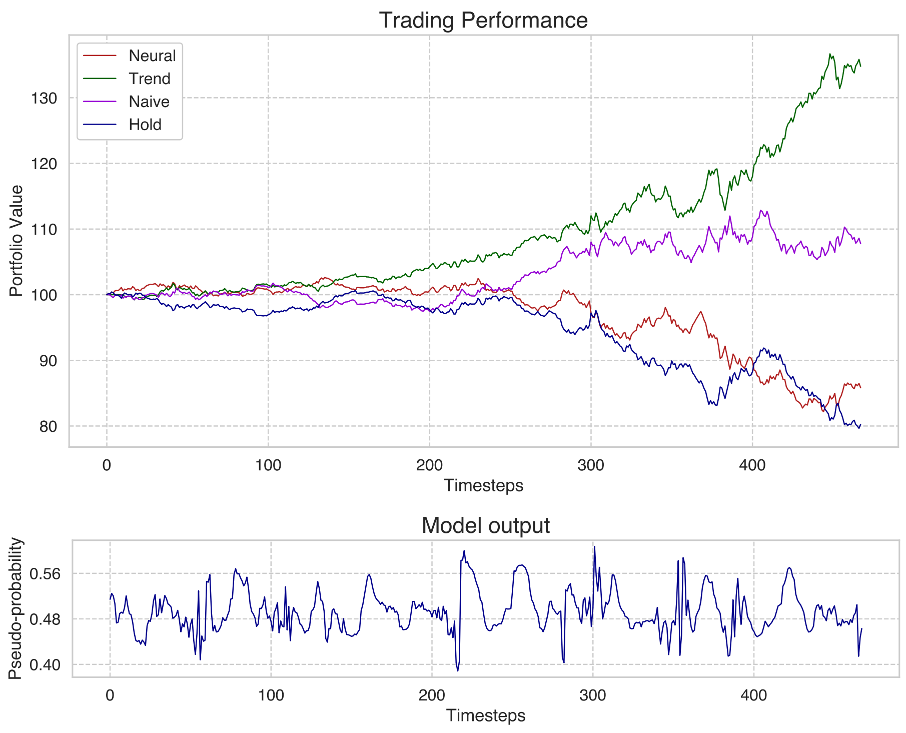

Figure A.1: Trading performance on the RX1 data set along with the neural network model's output, LSTM model.

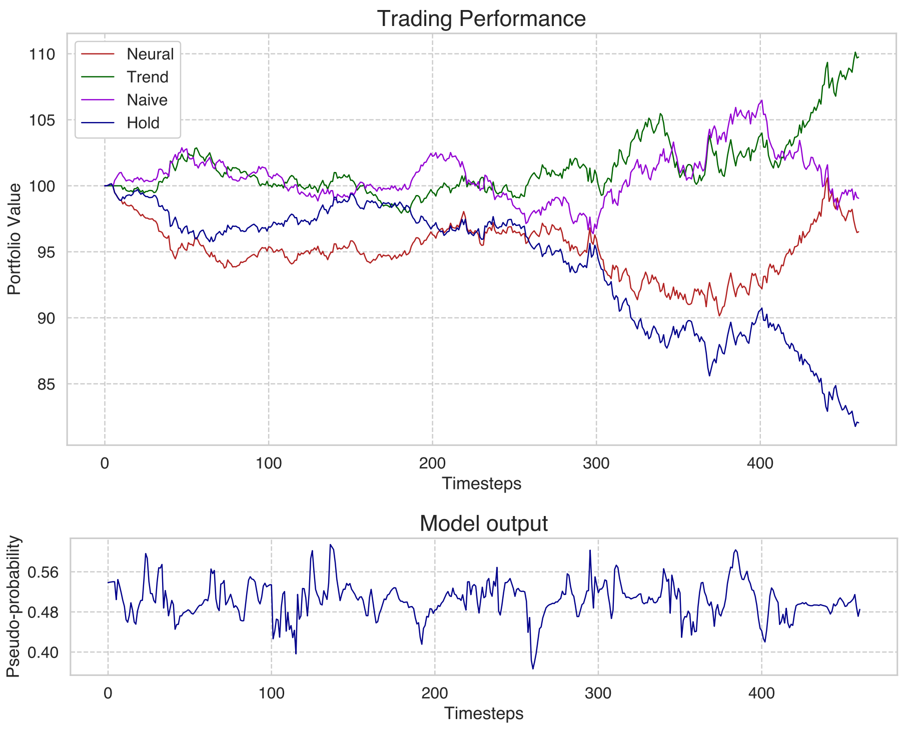

Figure A.2: Trading performance on the TY1 data set along with the neural network model's output, LSTM model.

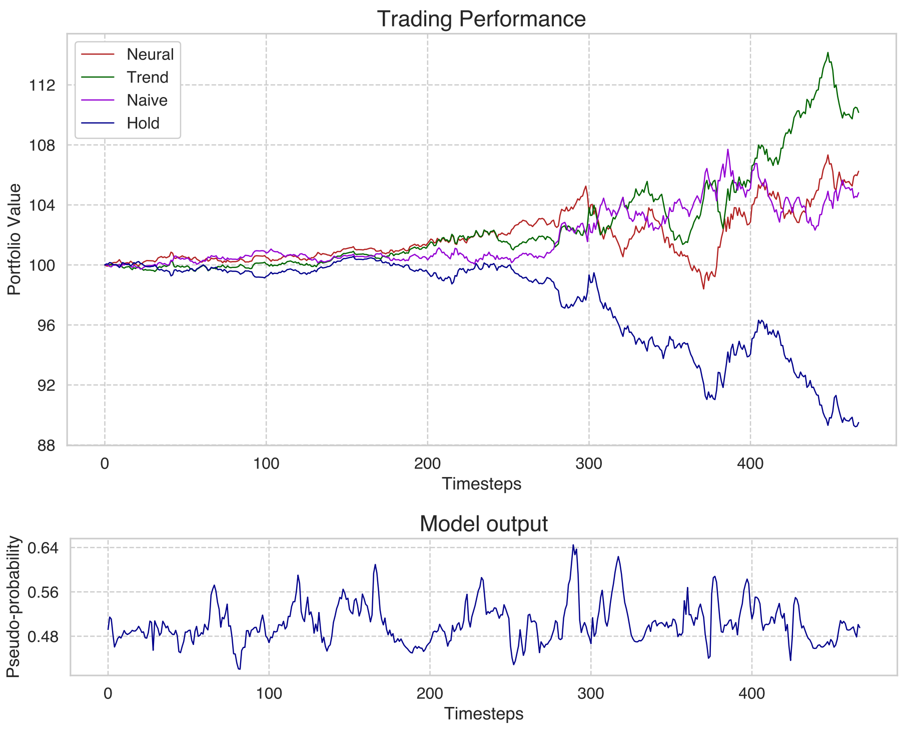

Figure A.3: Trading performance on the OE1 data set along with the neural network model's output, LSTM model.

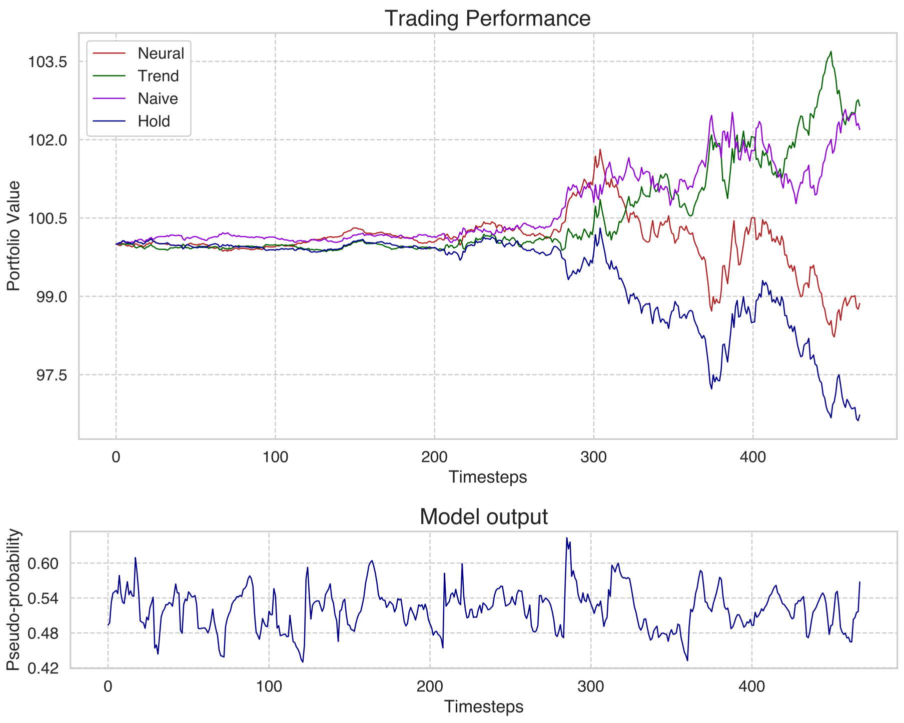

Figure A.4: Trading performance on the DU1 data set along with the neural network model's output, LSTM model.

A.1.2 GRU models trading performance

In this subsection the trading performance for the GRU models are displayed. Note that the Trend and Naive portfolio development is exactly the same as in the plots in section A.1.1; which is expected since these models are deterministic.

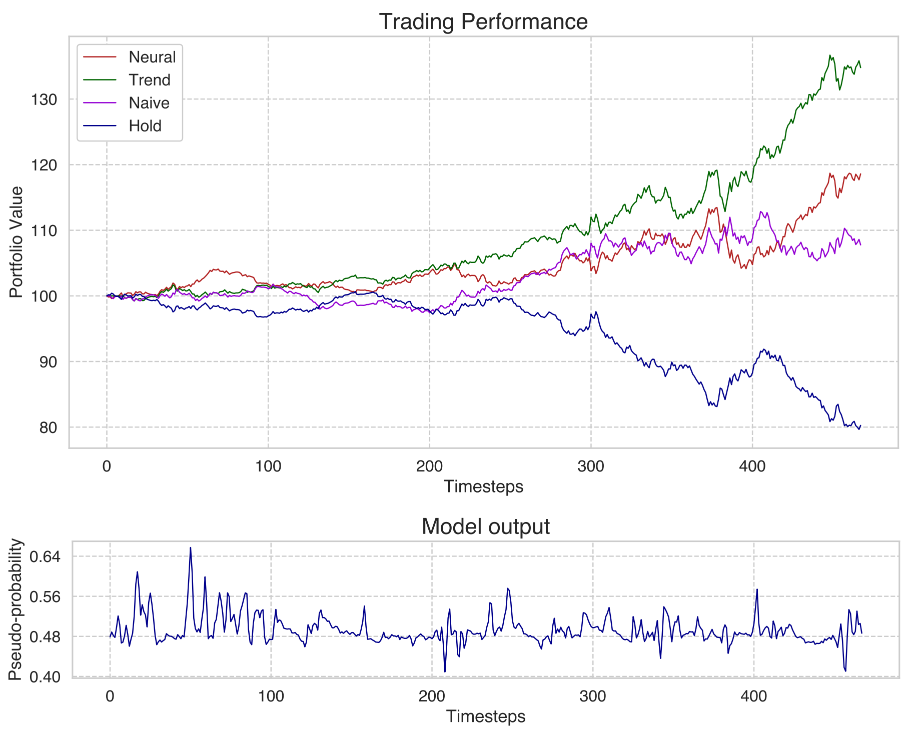

Figure A.5: Trading performance on the RX1 data set along with the neural network model's output, GRU model.

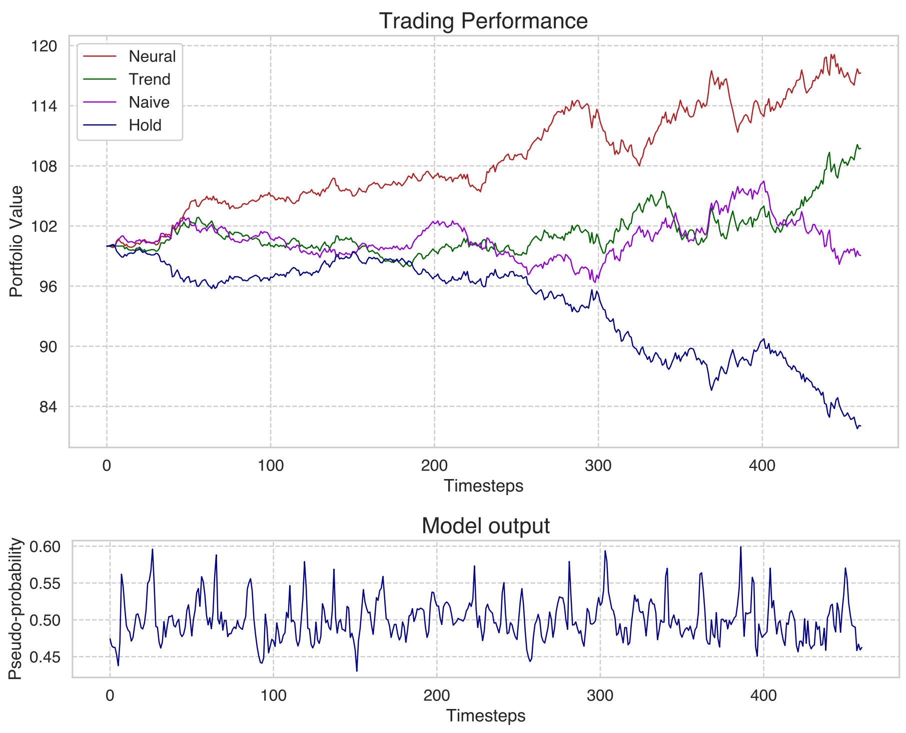

Figure A.6: Trading performance on the TY1 data set along with the neural network model's output, GRU model.

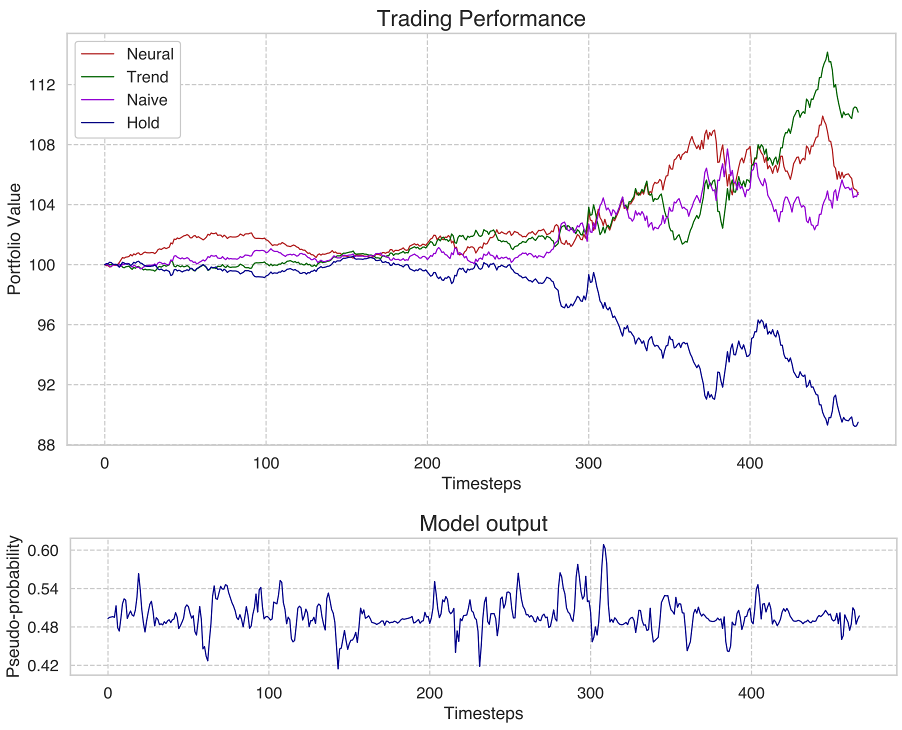

Figure A.7: Trading performance on the OE1 data set along with the neural network model's output, GRU model.

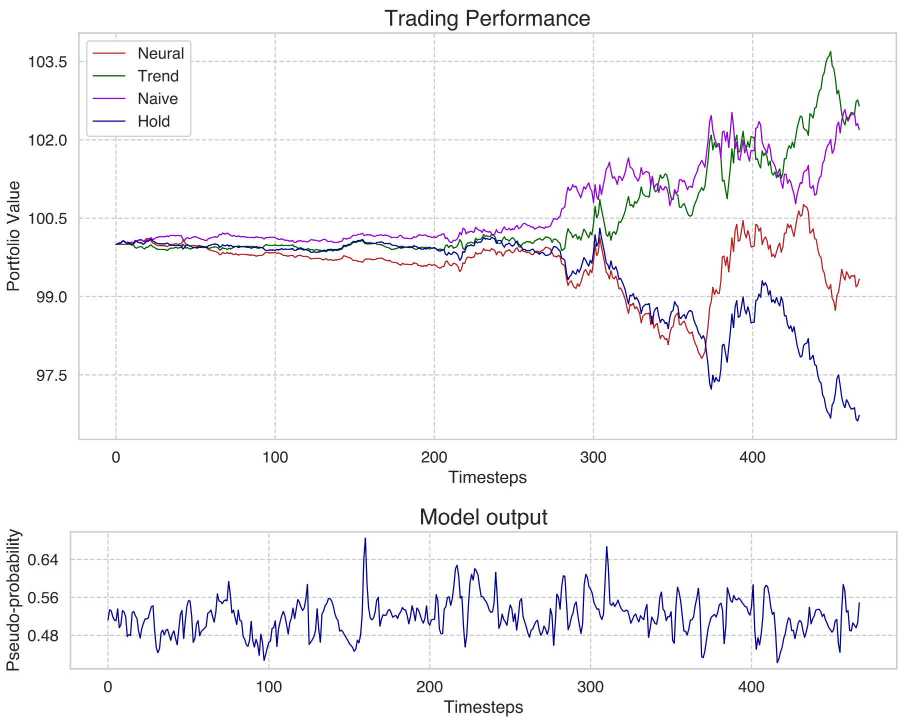

Figure A.8: Trading performance on the DU1 data set along with the neural network model's output, GRU model.

A.2 Benchmarking Results

Table A.1 presents benchmarking results for the financial metrics.

Table A.1: Financial benchmarking results.

Data	Model	Sharpe Ratio	Sortino Ratio	Gross Return (%)	Maximal Drawdown (%)
RX1	Lstm	(1.09)	(1.67)	(14.2)	(19.9)
	Gru	1.36	2.04	18.6	(8.22)
	Trend	2.29	3.73	34.8	(5.30)
	Naive	0.58	1.09	7.77	(6.66)
	Hold	(1.43)	(2.89)	(19.7)	(20.9)
TY1	Lstm	(0.29)	(0.53)	(3.48)	(9.99)
	Gru	1.41	2.70	17.28	(5.93)
	Trend	0.82	1.58	9.75	(5.09)
	Naive	(0.09)	(0.09)	(0.95)	(7.81)
	Hold	(1.61)	(3.00)	(17.9)	(18.3)
IK1	Lstm	1.90	3.20	25.2	(6.56)
	Gru	1.93	3.10	23.3	(8.29)
	Trend	1.43	2.40	17.8	(7.87)
	Naive	0.74	1.33	8.30	(7.81)
	Hold	(1.91)	(3.75)	(23.0)	(24.9)
OE1	Lstm	0.90	1.07	8.81	(5.59)
	Gru	(0.56)	(0.87)	(4.67)	(4.76)
	Trend	1.06	1.97	10.2	(3.98)
	Naive	0.72	1.05	4.81	(5.01)
	Hold	(1.23)	(2.40)	(10.5)	(11.3)
DU1	Lstm	(0.41)	(0.60)	(1.14)	(3.53)
	Gru	(0.20)	(0.36)	(0.67)	(2.30)
	Trend	0.82	1.36	2.65	(1.36)
	Naive	(0.80)	(1.13)	2.20	(1.71)
	Hold	(1.07)	(1.80)	(3.27)	(3.68)

In Table A.2 statistical benchmarking results are found.

Table A.2: Statistical benchmarking results.

Data	Model	True Positives	False Positives	True Negatives	False Negatives	Accuracy (%)
RX1	Lstm	66	101	158	143	47.9
	Gru	59	68	191	150	53.4
	Trend	92	81	172	113	57.6
	Naive	93	115	144	115	50.7
TY1	Lstm	114	132	109	106	48.4
	Gru	107	90	151	113	56.0
	Trend	73	86	149	143	49.2
	Naive	107	112	129	112	51.3
IK1	Lstm	45	56	102	69	54.0
	Gru	76	90	68	38	52.9
	Trend	35	45	106	76	53.8
	Naive	48	65	92	66	51.7
OE1	Lstm	104	117	140	107	52.1
	Gru	82	98	159	129	51.5
	Trend	89	94	157	118	53.7
	Naive	96	114	143	114	51.2
DU1	Lstm	173	159	64	72	50.6
	Gru	161	166	57	84	46.6
	Trend	87	85	134	152	48.2
	Naive	136	108	115	108	53.7

Table A.3 presents measures calculated from the statistical benchmarking results.

Table A.3: Measures calculated from the statistical benchmarking results. All values are in percentages.

Data	Model	True Positive Rate (%)	True Negative Rate (%)	Positive Predictive Value (%)	Negative Predictive Value (%)
RX1	Lstm	31.6	61.0	39.5	52.5
	Gru	28.2	73.8	46.5	56.0
	Trend	44.9	68.0	53.2	60.3
	Naive	44.7	55.6	44.7	55.6
TY1	Lstm	51.8	45.2	46.3	50.7
	Gru	48.6	62.7	54.3	57.2
	Trend	33.8	63.4	45.9	51.0
	Naive	48.9	53.5	48.9	53.5
IK1	Lstm	39.5	64.6	44.6	59.7
	Gru	66.7	43.0	45.8	64.1
	Trend	31.5	70.2	43.7	58.2
	Naive	42.1	58.6	42.5	58.2
OE1	Lstm	49.3	54.5	47.1	56.7
	Gru	38.9	61.9	45.6	55.2
	Trend	43.0	62.6	48.6	57.1
	Naive	45.7	55.6	45.7	55.6
DU1	Lstm	70.6	28.7	52.1	47.1
	Gru	65.7	25.6	49.2	40.4
	Trend	36.4	61.2	50.6	46.8
	Naive	55.7	51.6	55.7	51.6

In Table A.4 the correlation between the portfolio values resulting from the models and the underlying asset is found. Numbers in parenthesis means that it is a negative value.

Table A.4: Correlations with the underlying asset (Hold).

Data	Model	Correlation (%)	Volatility (%)
RX1	Lstm	88.4	7.26
	Gru	(95.2)	7.09
	Trend	(92.4)	7.61
	Naive	(83.1)	7.06
	Hold	100	7.80
TY1	Lstm	31.2	6.56
	Gru	(87.6)	6.44
	Trend	(84.0)	6.36
	Naive	(34.6)	6.09
	Hold	100	6.35
IK1	Lstm	(82.7)	12.2
	Gru	(67.4)	11.1
	Trend	(86.2)	11.4
	Naive	(86.4)	10.3
	Hold	100	11.3
OE1	Lstm	(58.8)	4.34
	Gru	(91.3)	4.44
	Trend	(88.1)	5.04
	Naive	(87.4)	3.58
	Hold	100	4.52
DU1	Lstm	63.4	1.50
	Gru	37.5	1.78
	Trend	(93.3)	1.74
	Naive	(85.9)	1.47
	Hold	100	1.66

A.3 Model Accuracy And Loss

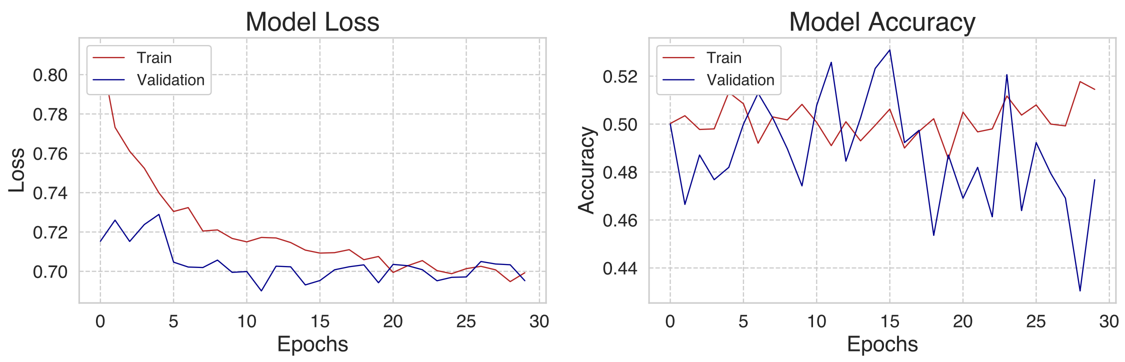

Figure A.9: The LSTM neural network model accuracy and loss on training and validation data, RX1 dataset.

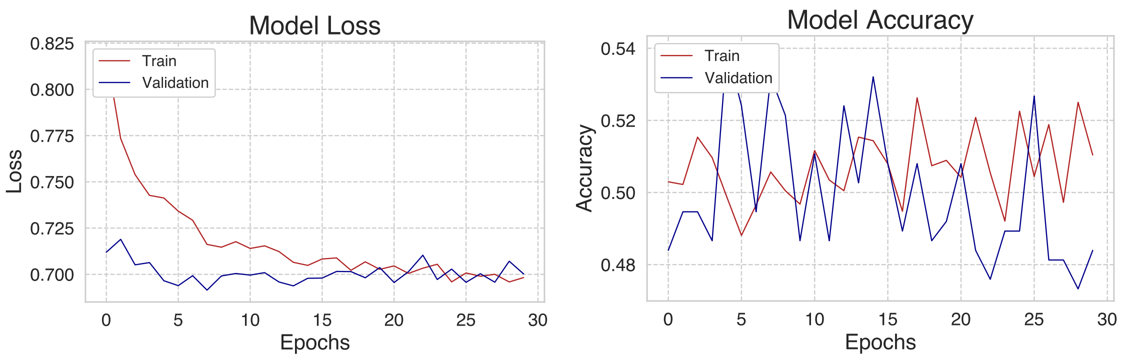

Figure A.10: The LSTM neural network model accuracy and loss on training and validation data, TY1 dataset.

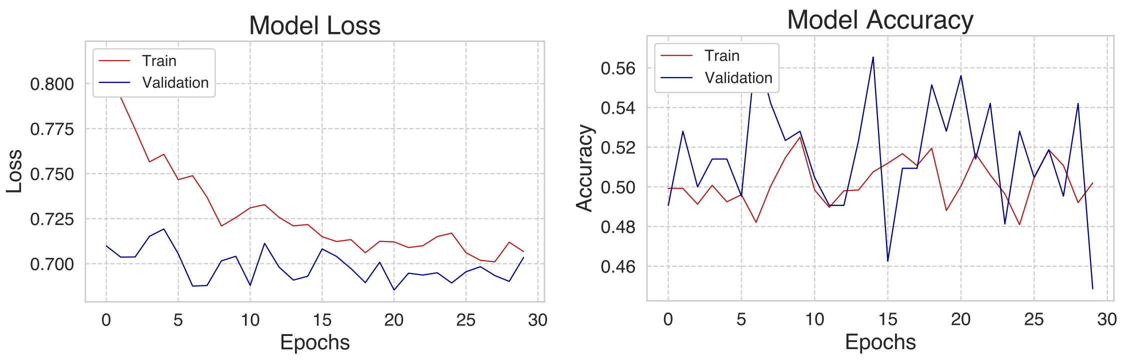

Figure A.11: The LSTM neural network model accuracy and loss on training and validation data, IK1 dataset.

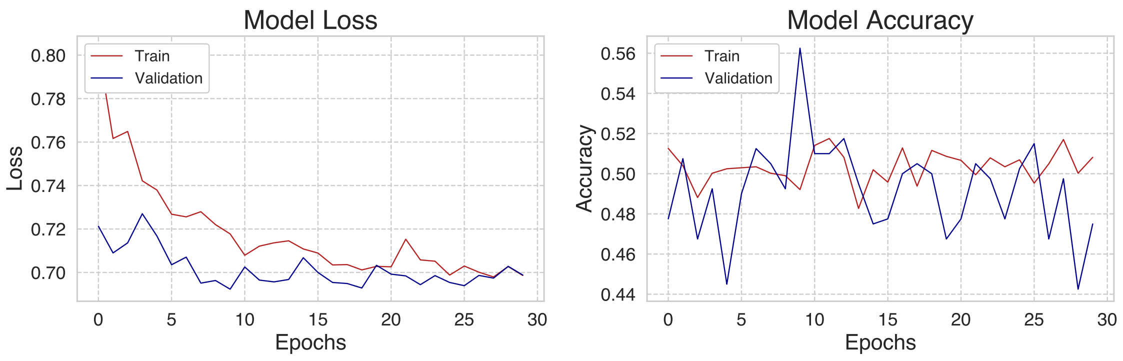

Figure A.12: The LSTM neural network model accuracy and loss on training and validation data, OE1 dataset.

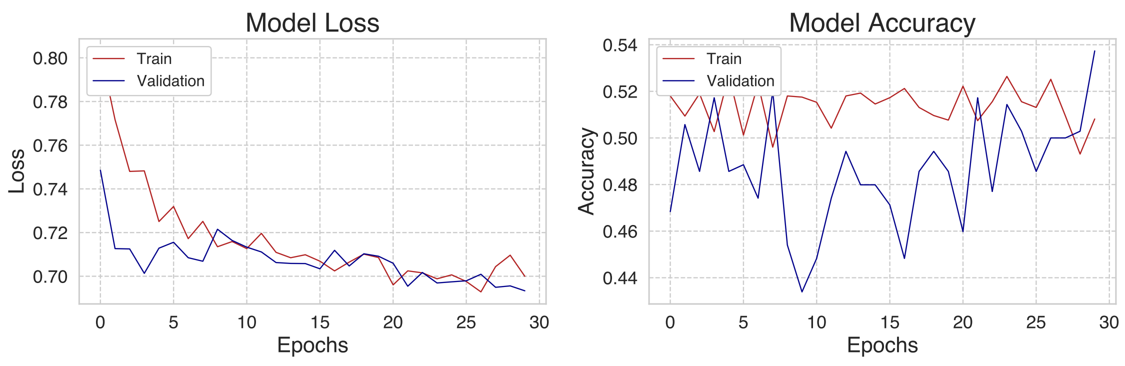

Figure A.13: The LSTM neural network model accuracy and loss on training and validation data, DU1 dataset.

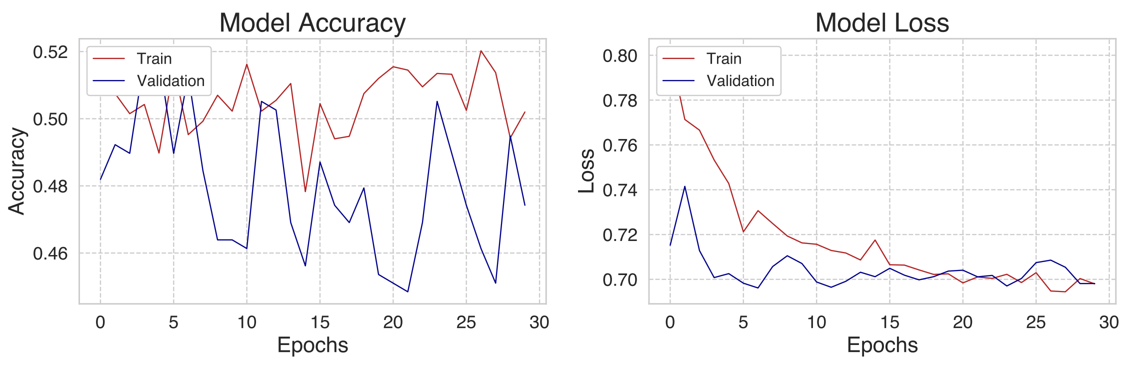

Figure A.14: The GRU neural network model accuracy and loss on training and validation data, RX1 dataset.

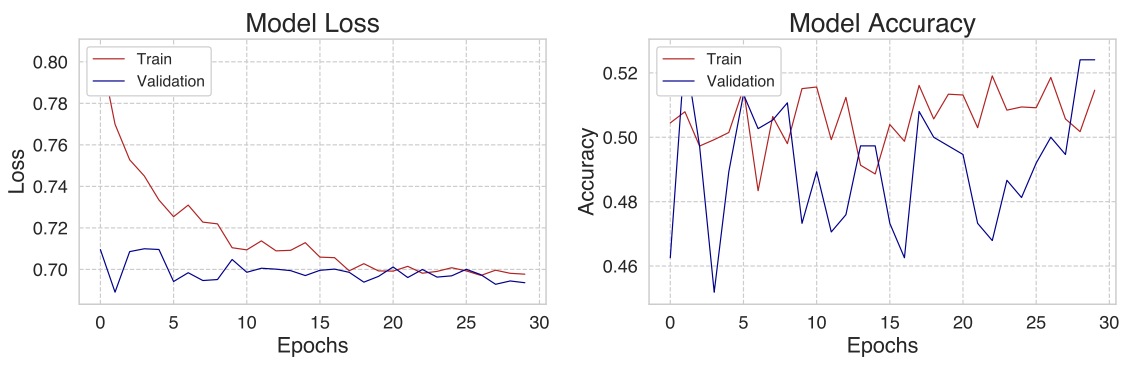

Figure A.15: The GRU neural network model accuracy and loss on training and validation data, TY1 dataset.

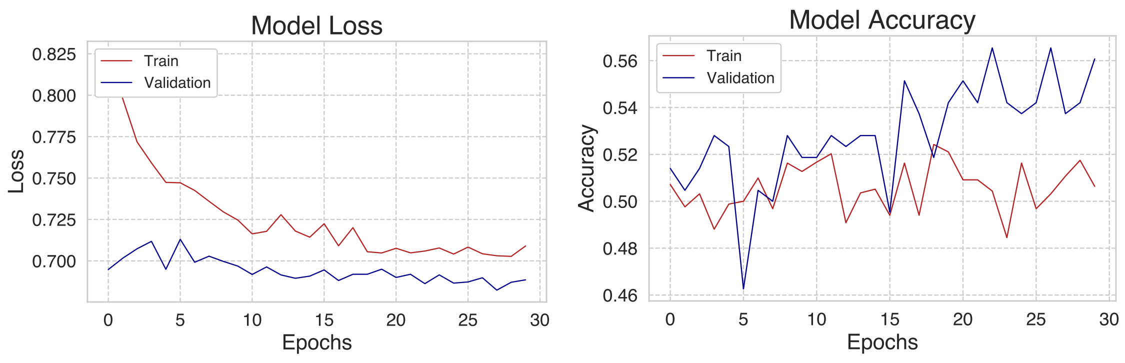

Figure A.16: The GRU neural network model accuracy and loss on training and validation data, IK1 dataset.

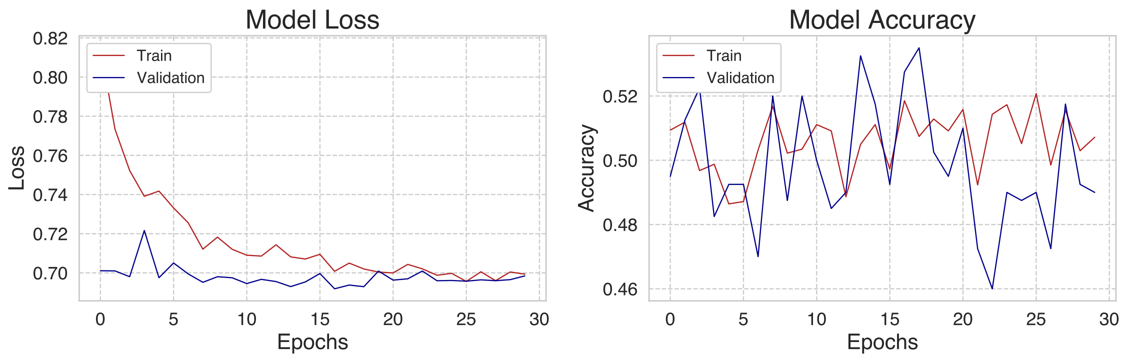

Figure A.17: The GRU neural network model accuracy and loss on training and validation data, OE1 dataset.

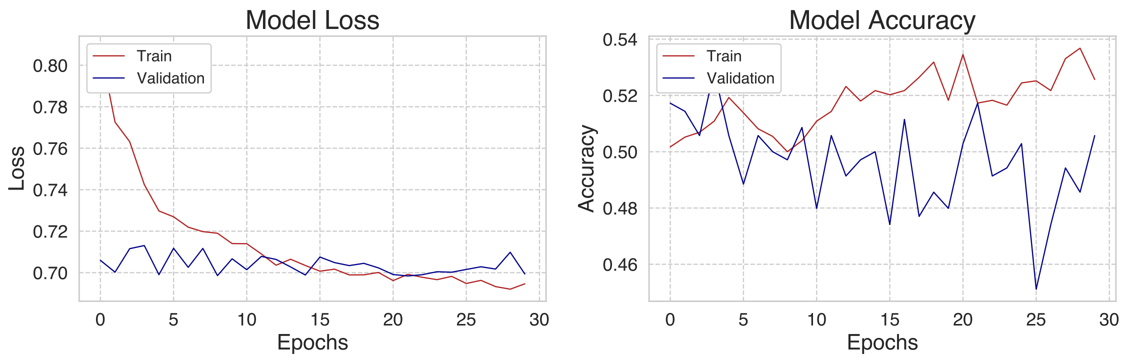

Figure A.18: The GRU neural network model accuracy and loss on training and validation data, DU1 dataset.