LUND UNIVERSITY

LTH
FACULTY OF ENGINEERING

# An Artificial Neural Network Approach to Algorithmic Trading

Timmie Bengtsson

January 2023

Master's degree in Financial Engineering

Examiner: Magnus Wiktorsson
Supervisors: Carl Hvafner & Erik Lindström

# Abstract

TODO———————————————————————
An abstract here...

This thesis was written in collaboration with Handelsbanken Asset Management as the concluding part of a master's degree at the Faculty of 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

TODO———————————————————————
Some acknowledgements...

# Contents


**1 Introduction** **1**

**2 Technical Background** **2**
2.1 Finance . . . 2
2.1.1 Long and short positions . . . 2
2.1.2 Trading bots and algorithmic trading . . . 2
2.1.3 Performance metrics . . . 3
2.2 Time Series Analysis . . . 3
2.2.1 Auto-regressive models . . . 3
2.2.2 Naive predictors . . . 3
2.3 Artificial Neural Networks . . . 3
2.3.1 Supervised Learning . . . 5
2.3.2 Recurrent neural networks . . . 5
2.3.3 Feature selection and extraction . . . 8
2.3.4 Imbalanced data . . . 9
2.3.5 Training neural networks . . . 9

**3 Methodology** **14**
3.1 Software . . . 14
3.2 Data . . . 14
3.2.1 Data storage and extraction . . . 14
3.2.2 Features . . . 15
3.2.3 Train-validation-test split . . . 15
3.2.4 Pre-processing . . . 15
3.3 Model Structure . . . 16
3.4 Trading Algorithms . . . 16
3.4.1 Time Series Analysis . . . 16
3.5 Benchmarks . . . 16

**4 Results** **17**
4.1 Benchmarks . . . 17
4.2 Cumulative Gross Returns . . . 18

**5 Discussion** **23**
5.1 Results . . . 23
5.2 Future work . . . 23

**References** **24**

**A** **27**

# List of Figures


2.1 Artificial neuron. . . . . 4
2.2 A simple neural network. . . . . 4
2.3 LSTM memory block. . . . . 7
2.4 A LSTM network. . . . . 8
2.5 Dropout. . . . . 11

4.1 Trading performance on the RX1 data set along with neural network model output. . . . . 19
4.2 Trading performance on the TY1 data set along with neural network model output. . . . . 20
4.3 Trading performance on the IK1 data set along with neural network model output. . . . . 21
4.4 Trading performance on the OE1 data set along with neural network model output. . . . . 22

A.1 RX1 neural network model's accuracy and loss on training and validation data. . . . . 27
A.2 TY1 neural network model's accuracy and loss on training and validation data. . . . . 27
A.3 TY1 neural network model's accuracy and loss on training and validation data. . . . . 28
A.4 IK1 neural network model's accuracy and loss on training and validation data. . . . . 28

# List of Tables

3.1 Data table . . . . . 14

4.1 Financial benchmarking results. . . . . 17
4.2 Statistical benchmarking results. . . . . 18
4.3 Measures calculated from the statistical benchmarking results. . . . . 18

A.1 Correlations with the underlying asset (Hold). . . . . 28

# 1

# Introduction

Machine learning methods have become an increasingly important tool for assisting with prediction, classification, and, most recently, generation. Neural networks seem especially promising when it comes to the task of forecasting time series data. Network architectures such as the LSTM and GRU usually perform as well as or better than traditional methods for times series data. This prompts one to question whether such models might also be able to predict asset prices.

This master thesis explores the use of artificial neural networks for predicting asset prices. We begin by reviewing the relevant literature on the use of neural networks for asset price prediction, and then discuss the specific algorithms that have been applied for this task.

Livieris et al. (2020) states that LSTM models can increase forecasting performance when predicting gold prices.

TODO———————————————————————

# 2

# Technical Background

## 2.1 Finance

Assuming that the reader is acquainted with basic financial concepts, and not to repeat basic concepts, we will keep this section brief.

### 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. When such a process is implemented by code in a computer it is commonly called a trading bot. When these trading bots then are deployed for trading on an exchange, it is sometimes 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

TODO———————————————————————
Economical: Sharpe-ratio, Sortino-ratio, Gross return, Max drawdown.
Statistical: 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).

## 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 $\varepsilon_t$ 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.

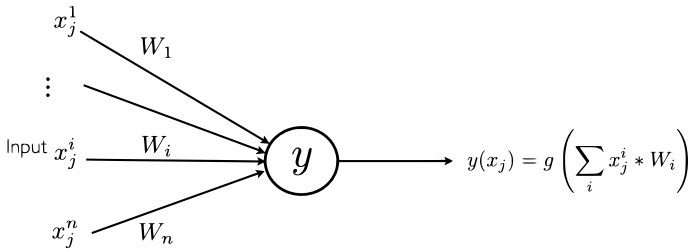



*Figure 2.1: Example of a single neuron network.*

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 (Graves, 2012)

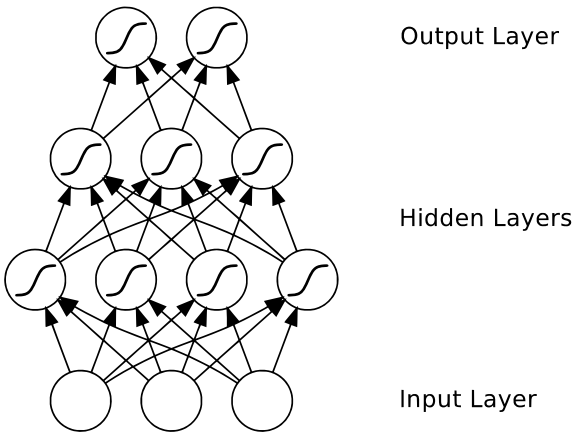



*Figure 2.2: The S-shaped curves in the hidden and output layers indicate the use of sigmoid activation functions.*

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, $\ell \in \{1, \ldots, L\}$, 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 $\ell$ 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 $\ell$, 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, $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_n, y_n)$, where $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n \in X$, $y_1, y_2 \ldots, y_n \in Y$ 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 $\phi_h$ and $\phi_y$ are activation functions (Amidi and Amidi).

The basic recurrent neural networks introduced by Elman in 1990 (Elman, 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). 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 (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.

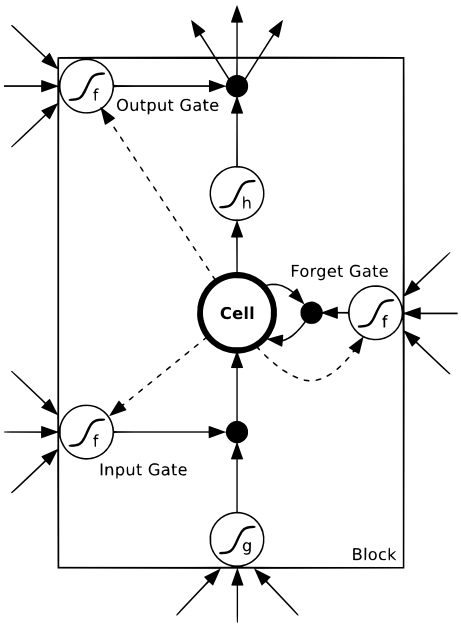



*Figure 2.3: A LSTM memory block with one cell.*

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.

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 $\phi$ 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.

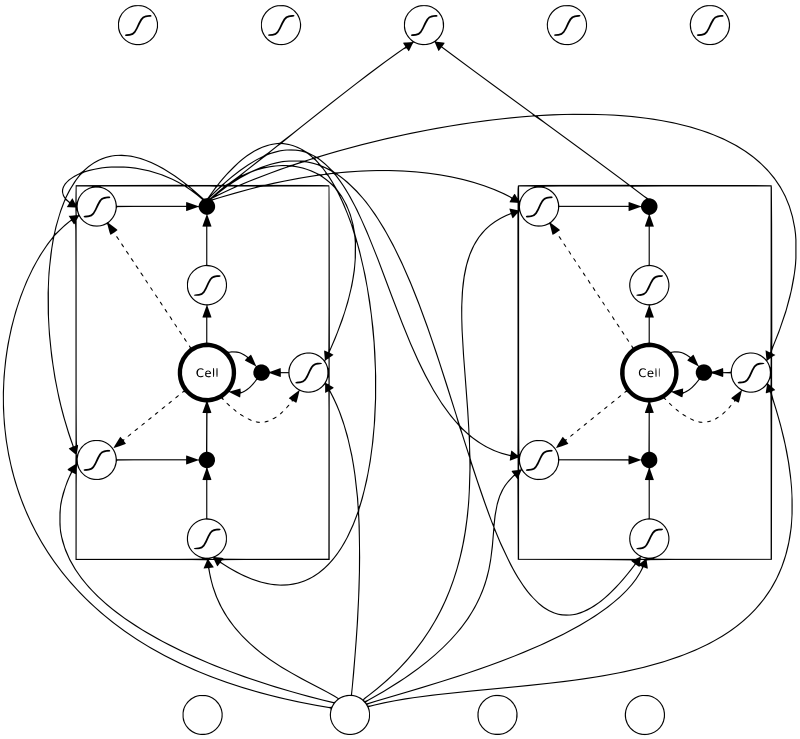



*Figure 2.4: A simplified LSTM network.*

**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 de-

sired 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),

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

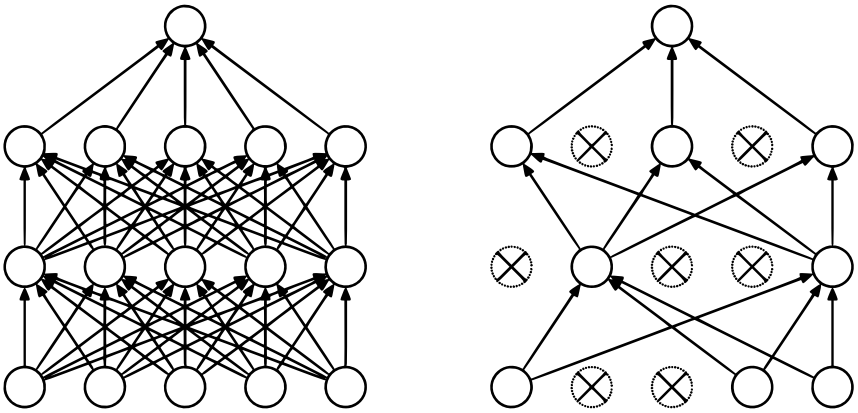
where $\eta$ is the learning rate.

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.

### 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.



(a) Standard Neural Net (b) After applying dropout.

*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.*

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: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

where $\epsilon$ 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 $\gamma$ and $\beta$ are learnable parameters, this allows $y_i$ to have any mean and standard deviation (Goodfellow et al., 2016).

### Binary cross-entropy

TODO———————————————————————

### 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 (Stanford University 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:** $W_t$

**procedure**
  $m_0 \leftarrow 0$
  $v_0 \leftarrow 0$
  $t \leftarrow 0$
  **while** $W_t$ not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_{W_t} f_t\left(W_{t-1}\right)$
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
    $\widehat{m}_t \leftarrow m_t / \left(1 - \beta_1^t\right)$
    $\widehat{v}_t \leftarrow v_t / \left(1 - \beta_2^t\right)$
    $W_t \leftarrow W_{t-1} - \eta \cdot \widehat{m}_t / \left(\sqrt{\widehat{v}_t} + \epsilon\right)$
  **return** $W_t$

where, $f(W)$ is the objective function with weights (parameters) $W$, $W_0$ are the initial weights, $\eta$ is the learning rate and $\beta_1$, $\beta_2$ and $\epsilon$ 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

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

where $\beta_1$ and $\beta_2$ 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 $\epsilon$ is needed to numerically stabilize the calculations, it is usually set to $10^{-8}$.

# 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 5 assets were recieved 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 after training. The results from the validation accuracy were used as a base to evaluate which model performed the best. In a iterative manner, the best model based on the validation data was saved and served as the baseline from which smaller changes were carried out. The improved model were then retrained on the training data and tested again on the validation data. When satisfied with the model, a final evaluation of the model's performance was done on the test data. This data is completely new to the model and has not been trained on or in any way incorporated into the model's creation.

### 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. To balance the data, the longer group 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 Model Structure

TODO———————————————————————
The models are neural networks with Long-Short-Term-Memory and Generalised-Linear-Unit structure.

The neural network model for each asset is trained on training data derived from that same asset.

## 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.

TODO———————————————————————
The AR(1) model is ..... AR(10) is...

## 3.5 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 then used to generate the Sharpe and Sortino ratios, and gross returns.

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

Table 4.1 presents benchmarking results for the financial metrics.

*Table 4.1: Financial benchmarking results.*

| Data | Model | Sharpe Ratio | Sortino Ratio | Gross Return | Maximal Drawdown |
|---|---|---|---|---|---|
| RX1 | Neural | (0.13) | (0.18) | (3.65)% | (12.9)% |
| | Trend | 2.42 | 3.42 | 34.8% | (5.30)% |
| | Naive | 0.47 | 0.74 | 7.77% | (6.66)% |
| | Hold | (2.07) | (3.23) | (19.7)% | (20.9)% |
| TY1 | Neural | (0.06) | (0.09) | 1.61% | (8.69)% |
| | Trend | 0.76 | 1.17 | 9.75% | (5.09)% |
| | Naive | (0.33) | (0.47) | (0.95)% | (7.81)% |
| | Hold | (2.35) | (3.39) | (17.9)% | (18.3)% |
| IK1 | Neural | (1.71) | (2.71) | (16.8)% | (19.4)% |
| | Trend | 1.41 | 2.19 | 17.8% | (7.87)% |
| | Naive | 0.65 | 1.11 | 8.30% | (7.81)% |
| | Hold | (2.41) | (3.97) | (23.0)% | (24.9)% |
| OE1 | Neural | 0.90 | 1.07 | 8.81% | (5.59)% |
| | Trend | 1.09 | 1.46 | 10.2% | (3.98)% |
| | Naive | 0.34 | 0.50 | 4.81% | (5.01)% |
| | Hold | (2.01) | (2.96) | (10.5)% | (11.3)% |
| DU1 | Neural | nbr | nbr | nbr% | maxdd% |
| | Trend | nbr | nbr | nbr% | maxdd% |
| | Naive | nbr | nbr | nbr% | maxdd% |
| | Hold | nbr | nbr | nbr% | maxdd% |

In Table 4.2 statistical benchmarking results are found.

*Table 4.2: Statistical benchmarking results.*

| Data | Model | True Positive | False Positive | True Negative | False Negative | Accuracy |
|---|---|---|---|---|---|---|
| RX1 | Neural | 72 | 90 | 169 | 137 | 51.5% |
| | Trend | 92 | 81 | 172 | 113 | 57.6% |
| | Naive | 93 | 115 | 144 | 115 | 50.7% |
| TY1 | Neural | 139 | 148 | 93 | 81 | 50.3% |
| | Trend | 73 | 86 | 149 | 143 | 49.2% |
| | Naive | 107 | 112 | 129 | 112 | 51.3% |
| IK1 | Neural | 67 | 97 | 61 | 47 | 47.1% |
| | Trend | 35 | 45 | 106 | 76 | 53.8% |
| | Naive | 48 | 65 | 92 | 66 | 51.7% |
| OE1 | Neural | 1 | 5 | 252 | 210 | 54.1% |
| | Trend | 89 | 94 | 157 | 118 | 53.7% |
| | Naive | 96 | 114 | 143 | 114 | 51.2% |
| DU1 | Neural | TP | FP | TN | FN | Acc% |
| | Trend | TP | FP | TN | FN | Acc% |
| | Naive | TP | FP | TN | FN | Acc% |

Table 4.3 presents measures calculated from the statistical benchmarking results.

*Table 4.3: Measures calculated from the statistical benchmarking results.*

| Data | Model | True Positive Rate | True Negative Rate | Positive Predictive Value | Negative Predictive Value |
|---|---|---|---|---|---|
| RX1 | Neural | 34.5% | 65.3% | 44.4% | 55.2% |
| | Trend | 44.9% | 68.0% | 53.2% | 60.3% |
| | Naive | 44.7% | 55.6% | 44.7% | 55.6% |
| TY1 | Neural | 63.2% | 38.6% | 48.4% | 53.5% |
| | Trend | 33.8% | 63.4% | 45.9% | 51.0% |
| | Naive | 48.9% | 53.5% | 48.9% | 53.5% |
| IK1 | Neural | 58.8% | 38.6% | 40.8% | 56.5% |
| | Trend | 31.5% | 70.2% | 43.7% | 58.2% |
| | Naive | 42.1% | 58.6% | 42.5% | 58.2% |
| OE1 | Neural | 0.47% | 98.1% | 16.7% | 54.6% |
| | Trend | 43.0% | 62.6% | 48.6% | 57.1% |
| | Naive | 45.7% | 55.6% | 45.7% | 55.6% |
| DU1 | Neural | TPR% | TNR% | PPV% | NPV% |
| | Trend | TPR% | TNR% | PPV% | NPV% |
| | Naive | TPR% | TNR% | PPV% | NPV% |

## 4.2 Cumulative Gross Returns

It might be informative to observe how trading returns develop over time for each model. Therefore, this section contains figures displaying this performance for the different assets. The performance of the neural network model (Neural) for each asset is displayed. This performance is seen along with the benchmark models

(Trend and Naive) performance 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 Appendix A.

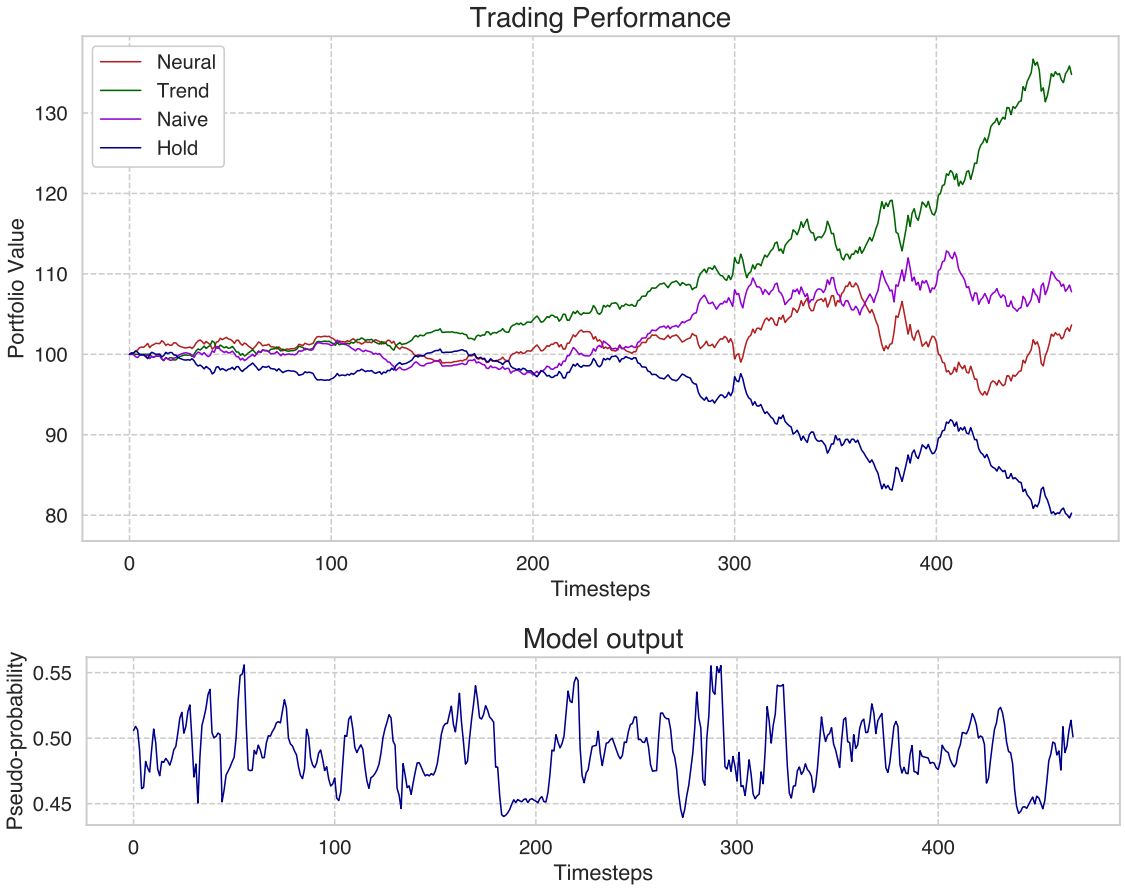



*Figure 4.1: Trading performance on the RX1 data set along with the neural network model's output.*

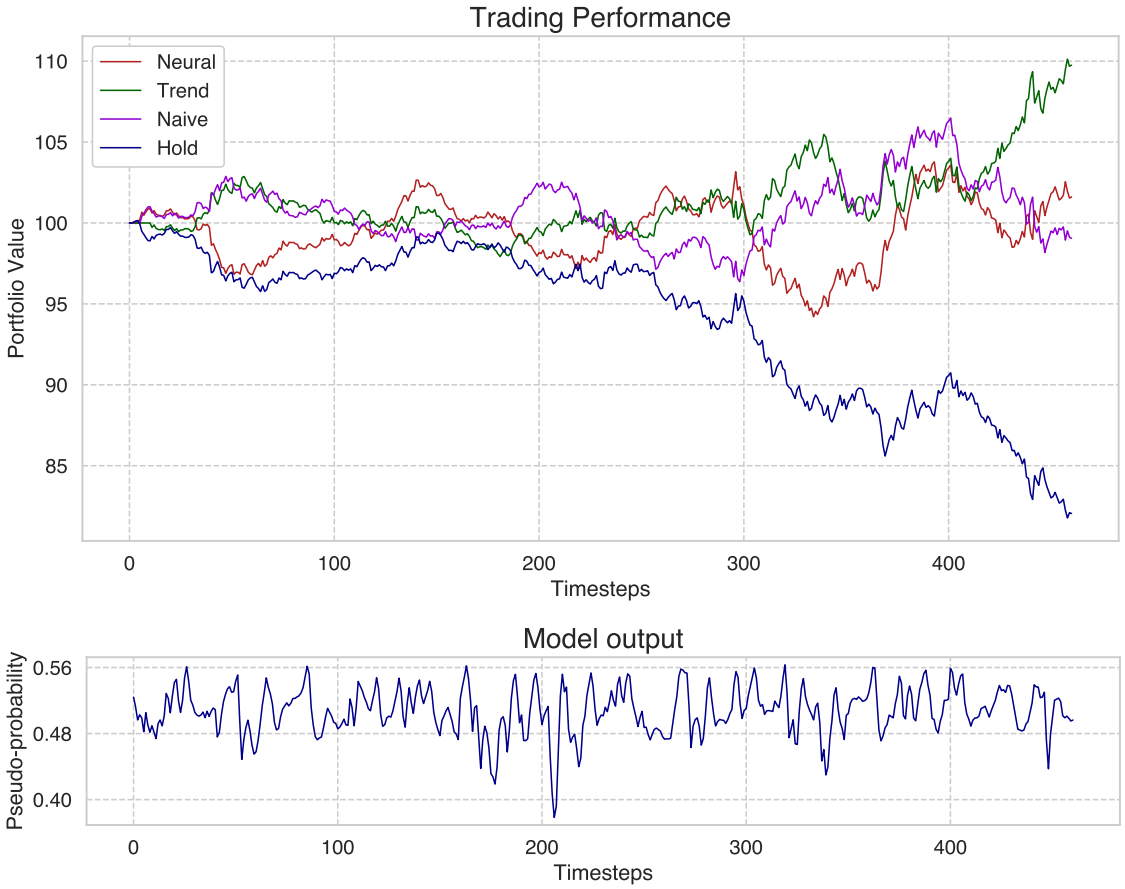



*Figure 4.2: Trading performance on the TY1 data set along with the neural network model's output.*

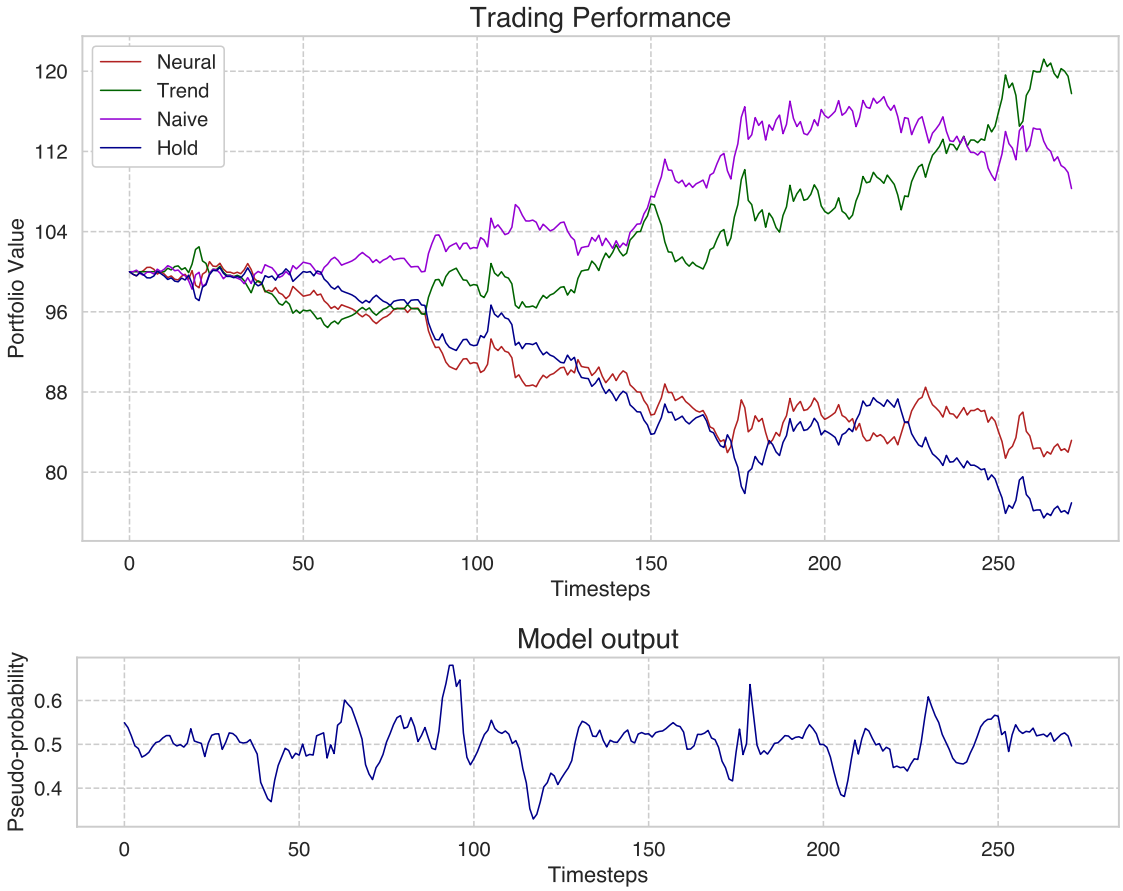



*Figure 4.3: Trading performance on the IK1 data set along with the neural network model's output. Note that the number of timesteps (trading days) are fewer in this data set.*


*Figure 4.4: Trading performance on the OE1 data set along with the neural network model's output.*

TODO———————————————————————

Pics and some comments

# 5

# Discussion

## 5.1 Results

TODO———————————————————————

There is a possibility that the constant validation loss observed is an artifact of essentially trying to fit on noise, or something perceived as noise due to, for example, an extremely high level of complexity in the signal. In such a case, any, even seemingly plausible, predictions the model emits may be instead be spurious. Assuming that this is true, one might want to try extracting more informative features, using a larger variety of features, or performing more data augmentation on the training set.

OUTCAST: Drawdowns are usually more servere, that is, there are alot of small up days and few down days but the down days are often larger. IThis is not (?) incorporated into the model. If so it would try to capture the down days and in doing so sacrifices accuracy, which might be a desirable dynamic to capture in the model. If not, this might be an idea worth trying to evaluate and implement.

OUTCAST: The models need to be evaluated on a lot more examples and series in order to determine efficacy. Now the naive and trend outperform due to the underlying trending down quite smoothly. Or a simple trending model might actually be an good idea...? Horsetrading HongKong linear regression worked wonders. And so do simple momentum strategies as well "insert fact from historical performance" and still will work according to some financial figures "insert reference to this".

OUTCAST: It seems that the naive models do perform quite well compared to the neural model across all measures, Implying that the neural model is ineffective.

## 5.2 Future work

TODO———————————————————————

OUTCAST: How does the algos perform if treshholds are set. prob above or under some value in order to take a decision, else do nothing? Evaluate the predictions the day before larger movements, are the predictions in any way indicating that something bigger might happen the next day as compared to the other predictions?

Train on multiple different assets in order to predict one of them? Or use multiple assets as inputs for the prediction of one of them. Implement macrodata (or similar)? How to do this in a reasonable manner fitting the neural network structure? Develop models for shorter time frame data, like 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 this market dynamics and train another model to identify what dynamics are prevaling given some input data? And can outputs from this side-model then be used as inputs to the trading algo? Implement some information about the perfromance from very long ago, maybe less and less info but thematically accurate so that the model have some sense of it the recent few years have been very positive or negative. Design smarter evaluation of performance, reinforcement learning, I think personally believe this is one of the approaches with the highest probability of success. Can continiously train the model new data to tune it using the latest data? Weight the allocation/trade size depending on how certain the model is in a trade (as deemed by how much the output deviates from 0,5).

Maybe exclude this section..? do if there's time...

# Bibliography


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

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.

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

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.

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. 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.

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

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.

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

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.

Stanford University CS231n. Cs231n convolutional neural networks for visual recognition. URL `https://cs231n.github.io/neural-networks-3/`.

# Appendix A

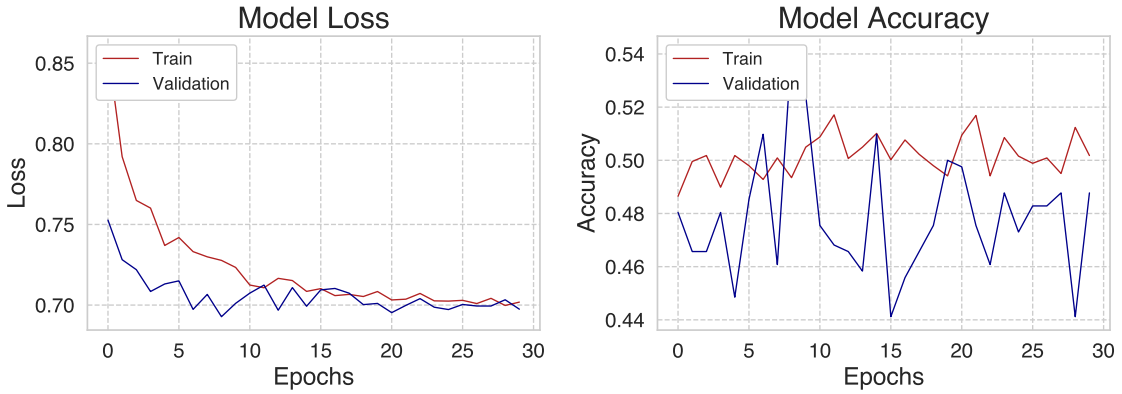



*Figure A.1: The RX1 neural network model accuracy and loss on training and validation data.*

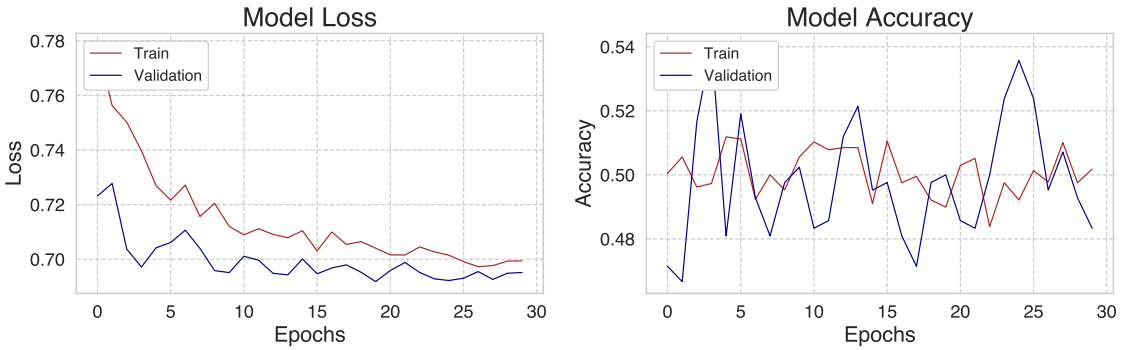



*Figure A.2: The TY1 neural network model accuracy and loss on training and validation data.*

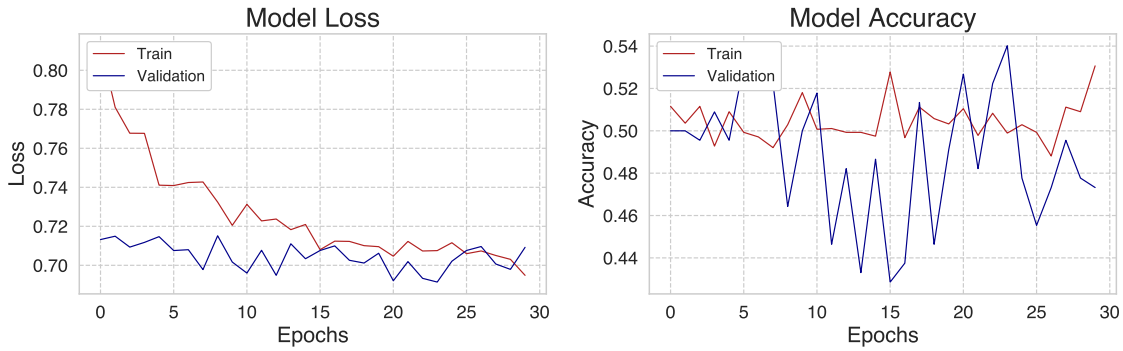



*Figure A.3: The TY1 neural network model accuracy and loss on training and validation data.*

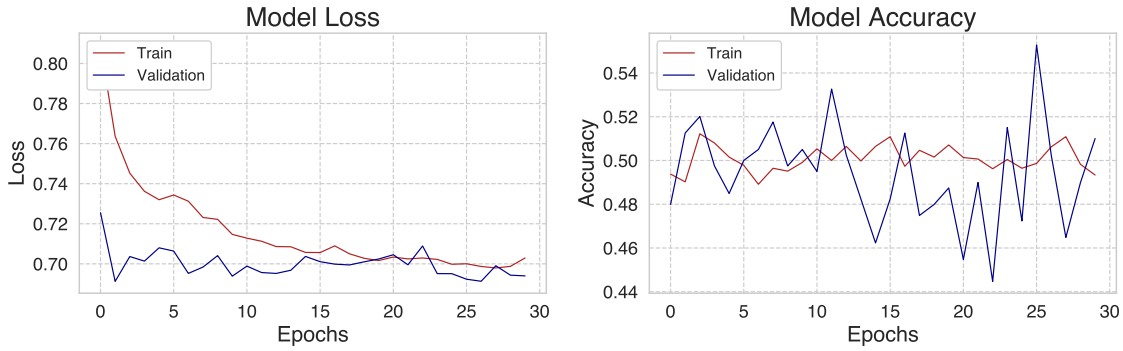



*Figure A.4: The IK1 neural network model accuracy and loss on training and validation data.*

In Table A.1 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.1: Correlations with the underlying asset (Hold).*

| Data | Model | Correlation |
|---|---|---|
| RX1 | Neural | (18.5)% |
| | Trend | (92.4)% |
| | Naive | (83.1)% |
| TY1 | Neural | (5.95)% |
| | Trend | (84.0)% |
| | Naive | (34.6)% |
| IK1 | Neural | 93.8% |
| | Trend | (86.2)% |
| | Naive | (86.4)% |
| OE1 | Neural | (98.7)% |
| | Trend | (88.1)% |
| | Naive | (87.4)% |
| DU1 | Neural | 000% |
| | Trend | 000% |
| | Naive | 000% |