DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Application of Machine Learning to Financial Trading

MICHAL HOREMUZ

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Application of Machine Learning to Financial Trading

MICHAL HOREMUZ

Master in Computer Science
Date: June 30, 2018
Supervisor: Mårten Björkman
Examiner: Örjan Ekeberg
School of Electrical Engineering and Computer Science

Abstract

Machine learning methods have become powerful tools used in multiple industries. They have been successfully applied to problems such as image recognition, speech recognition and machine translation, among others. In this report, we investigated several machine learning methods for forecasting five different bond indexes. We have implemented and analyzed Feedforward Neural Nets, LSTMs, Q-Networks and Gradient Boosted Trees, and compared them to the Buy&Hold strategy. We performed manual feature extraction based on some popular features used in the industry. The features were extracted from several financial instruments and were used as predictor variables. The results showed that XGBoost and Feedforward Neural Networks were consistently able to beat the Buy&Hold strategy for three of five bond indexes.

Sammanfattning

Maskininlärningsmetoder har blivit kraftfulla verktyg som används i flera problemområden. De har framgångsrikt tillämpats på problem som bland annat bildigenkänning, taligenkänning och maskinöversättning. I denna rapport har vi undersökt flera maskininlärningsmetoder för att förutse fem olika obligationsindex. Vi har implementerat och analyserat Feedforward Neural Nets, LSTMs, Q-Networks och Gradient Boosted Trees, och jämfört dem med Buy&Hold strategin. Vi har utfört manuell extraktion av features baserat på några populära funktioner som används inom industrin. Dessa features beräknades från flera finansiella instrument och användes som prediktorvariabler. Resultaten visar att XGBoost och Feedforward Neural Networks kan konsekvent slå Buy&Hold strategin för tre av fem obligationsindex.

Acknowledgements

I want express my gratitude towards Mårten Björkman, my supervisor at KTH. Thank you for the advice, feedback, and guiding me through this project.

I am also grateful towards Marek Ozana and Anders Nordborg from Excalibur Asset Management AB. You have both shared a lot of knowledge with me, while showing me a lot of patience. Your help has been very appreciated, and this project would not be possible without you.

Contents

Chapter 1

Introduction

AI is making its way to financial markets. In the search for uncorrelated strategies hedge fund managers are increasingly adopting quantitative strategies. Beyond the "traditional" quantitative strategies, a new source of competitive advantage is emerging with the application of Machine Learning tools to analyze financial data. Machine learning may detect complex patterns that traditional quantitative analysis is not able to identify. Machine Learning strategies have multiple advantages over humans in financial trading. Perhaps the most important advantage is they have no emotions; they do not feel fear during a crash nor euphoria when markets are up. Another advantage is being able to consider multiple features from multiple time series simultaneously.

The purpose of this paper is to investigate several machine learning algorithms and compare their performance to the industry standard buy-and-hold strategy. Performance is measured by Sharpe Ratio. The focus of this study is forecasting various bond indexes.

The financial assets considered in this paper can be seen as time series, where each point in the series is the price of the asset at a certain time. Correctly forecasting the price, or movement of the price, allows investors to profit by buying when the price is predicted to rise, and sell when it is predicted to fall. This has very real practical applications and is is of very high interest to not only funds, but also private investors.

1.1 Research Question

Can Machine Learning methods such as Neural Networks, and LSTMs, Boosted Trees and Q-Networks be trained to predict financial time series data, and outperform the industry standard "buy and hold" strategy? Specifically, we are interested in predicting bond indexes.

1.2 Ethical Considerations

Using machine learning to automate speculative trading has little or no ethical ramifications. One could argue about the ethical implications of having quantitative trading, when a group of expert traders have access to much better tools than others.

On the other hand, with development in machine learning, even small individual traders might get access to tools comparable to those that large traders already have.

One benefit from speculative trading in general is making markets more efficient by moving prices of assets to closer reflect their real value. Another benefit is adding liquidity to markets. With machine learning models, there is also a risk that given enough resources, a model can begin to influence the market. Market manipulation is illegal, and when an AI does it it can have unpredictable and potentially dangerous effects on society.

1.3 Related Works

There exists numerous papers and other literature that consider the problem of financial time series forecasting. Algorithms based on Support Vector Machines, (Deep, Recurrent) Neural Networks and Decision Trees are common in literature.

Kolanovic and Krishnamachari (2017) [1] from J.P Morgan discuss and compare many algorithms as well as useful features and preprocessing techniques. They also provide an introduction to many algorithms, such as Support Vector Machines (SVM) Artificial Neural Networks (ANN), Convolutional Neural Networks (CNN) [2], Long Short-Term Memory networks (LSTM) [3] and XGboost [4]. Several features and algorithms that they describe are also used in this report.

Although they have not tested all mentioned algorithms, their results indicate that XGboost gives the best performance.

Another study from J.P Morgan by Salem et al. (2017) [5] compared multiple "classic" machine learning methods. They also used Principal Component Analysis as a dimensionality reduction tool for their feature set. They concluded that Random Forests [6] showed "the most promise".

One more important paper was by Ungari, Ramegowda and Turc (2013) [7], where the authors use a SVM for forecasting several different stock indexes. One very interesting thing about the study is their cross validation method, which has been adapted for this report.

Longmore (2016) has posted two blogs [8] [9] on this subject which are very informative. Longmore describes a general process for financial forecasting. His first blog [8] is concerned with different feature types, and several feature selection processes. In this report, several features have been adapted from Longmore. In the second blog [9] Longmore tests several models with his features, where the results show that ANNs perform the best. Interestingly, his results show Random Forests as being the worst, which is in contrast to Salem et al. (2017)[5].

A Deep Neural Network (DNN) was tested by Dixon, Klabjan and Bang (2016) [10]. The tests were run on higher frequency data: 5 minute interval futures prices. Their network was trained to classify the next value in the time series into 3 categories: negative, flat, and positive price movement. They use a fairly large network with an excess of twelve million weights. Their results show high accuracies for several commodities.

There have not been many attempts to use reinforcement learning in this field. One example is the paper by Lu (2017) [11]. There the author makes use of a LSTM network to find a trading strategy for the GBPUSD exchange rate. The results are positive, with a Sharpe ratio of 0.12. In this paper we implement and analyse a Deep Q network, based on the original paper by Mnih et al. (2013) [12].

1.4 Background

This section is dedicated to giving a brief review of terminology and some background knowledge.

1.4.1 Bonds

When a company or other entity requires money, they may either borrow money from a bank, or issue so-called bonds. A bond is a contract to pay back the loan with a certain interest at a certain time. The amount of money the issuer is borrowing is called the "face value" of the bond, the interest rate is the "coupon", and the time at which the issuer is to repay the loan is the "maturity date". When bonds are issued, an investor can buy a bond at face value, thereby loaning the money to the issuer. Bonds can be bought and sold by investors on the market, where the market price is determined by several factors, including the coupon rate, the maturity date, and the credit rating of the issuer.

Credit rating is a score given to entities such as companies based on their perceived risk of default. The higher the risk, the lower the rating. Bonds are categorized into several credit rating classes by entities such as Standard & Poor's. The higher ranked bonds are referred to as "investment grade", and the lower ranked ones are referred to as "junk bonds" or "high yield bonds". Lower ranked bonds typically have higher coupon rates to offset their higher risk of default.

The market price of a bond makes up a time series. A weighted average of the prices of a collection of bonds (or any other assets) is called an index. In this report, we try to predict if the price of several indices will rise, or fall.

1.4.2 Trading positions

The terms long and short refer to trading positions. In simple terms, a "long position" or "going long" means you are betting for the market price to rise, while inversely a "short position" or "going short" means you are betting for the price to fall.

In practice, going long means you buy an asset and hope for the price to rise, and therefore make a profit when you sell. Example:

You predict that the price of shares of company A will rise. You buy some shares of company A at a price of 10 units. Some time passes and the market price for your shares has risen to 11 units. You sell at the market price and make a profit of 1 unit.

Of course, the price could fall, and you lose money. Note that the example is somewhat incomplete due to missing details such as transaction fees. We omit these details for now.

A short position in practise is a little different from the long position. Example:

You predict that the price of shares of company A will fall. You borrow 10 units worth of shares and sell them at the market price. You get 10 units, but you owe the shares to the lender. Some time passes and the market price of the owed shares drops to 9 units. You buy the shares back at a price of 9 units and return them to the lender, making a profit of 1 unit.

Again some details are omitted, such as the lending fee.

Note that due to the fact that the price of an asset can not go below zero, you can not lose more money than what you have invested in the long position. However because the price of an asset can rise almost unbounded, you can suffer heavy losses from a short position.

In this report, for the sake of calculating performance, when we predict the price of an asset will rise we will go long, and when we predict the price to fall we will go short.

1.4.3 Market Efficiency

Market efficiency is a measure of how well prices reflect all available information. The prices in an efficient market completely reflect the value of the asset being traded, making it impossible to buy undervalued assets, and sell overvalued ones. The Efficient Market Hypothesis (EMH) was first introduced by Fama (1970) [13] and it states that due to market efficiency it should be impossible to outperform the market consistently over long periods of time. The hypothesis assumes that everyone has all relevant information and that everyone acts with some degree of rationality. There has been much discussion relating to the validity of the EMH, which is out of scope of this paper.

Chapter 2

Algorithms

2.1 Feedforward Neural Network

Artificial Neural Networks are inspired by the neurons in human brains. Each neuron receives inputs from other neurons and depending on the inputs, it "fires" a signal to other connected neurons. In a Feedforward Neural Network (FFNN), the "neurons" are arranged in layers, where the inputs are fed into one end, and the output is received on the other end, see Fig. 2.1. All neurons of a previous layer are connected to all other neurons of the next layer. These layers are known as "fully connected" layers.

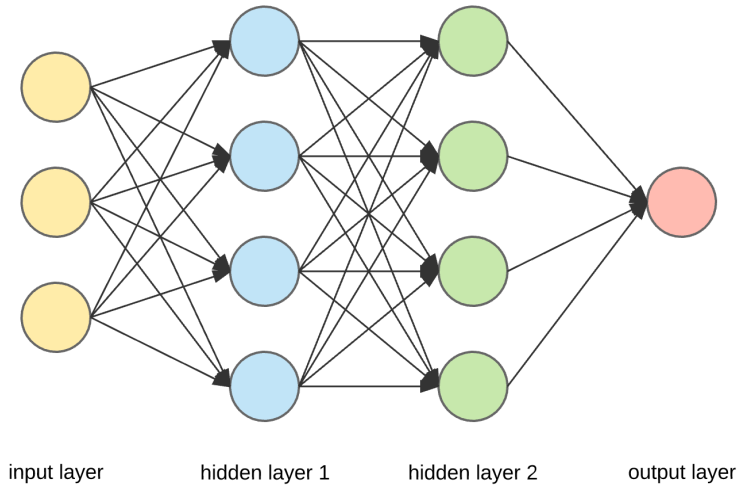


Figure 2.1: A FFNN with two hidden layers.

Let the input be a vector $x \in \Re^n$. For a FFNN with two hidden

layers, the output would then be:

$$\begin{aligned} h^1(x) &= f^1(W^1 x + b^1) \\ h^2(h^1) &= f^2(W^2 h^1 + b^2) \\ y_p(h^2) &= f^3(W^3 h^2 + b^3) \end{aligned} \tag{2.1}$$

where h^i are the outputs from each hidden layer, and y_p is the output of the network. W^i are weight matrices of size $m_i \times m_{i-1}$, where m_i is the number of neurons in layer i. Each element of W^i represents a weight from a neuron at layer $i-1$ and layer i, in other words the arrows in Fig 2.1. The vectors b^i are called bias vectors, and are of size m_i. Together, the expression $W^i h^{i-1} + b^i$ give a vector of size m_i where each element is the sum of all activation of the previous layer to a neuron in the current layer, plus a bias term. Each element is then fed into a function f^i called an "activation function". There are many choices for activation function, but the main idea is to act like a (non-linear) threshold for when the neuron should "fire" (or activate) given the sum of activations of connected neurons.

2.1.1 Activation functions

There are many different possible activation functions, but in this report we consider only the "relu" and "tanh" and "sigmoid" functions. The "tanh" activation function is:

$$h(x) = tanh(x) \tag{2.2}$$

In this report it is only used as an activation function for the final layer. This has the effect of "squeezing" the final output of the network to the range -1 to 1. Note that the gradient disappears for large and small values of x.

The "sigmoid" functions is given by:

$$h(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

The function behaves almost the same as the "tanh" functions, except it scales values to between 0 and 1. It is also only used as the output layer activation function in this report.

The "relu" activation function, also known as the rectified linear unit is:

$$h(x) = max(0, x) \tag{2.4}$$

This simple activation function has been demonstrated to be superior to the previously used sigmoid function [14]. Advantages include an easy to calculate gradient, and that the gradient does not saturate for high input values, which reduces vanishing gradient problems with deeper networks. In this report it is used for all hidden layers.

2.1.2 Loss function

When training a FFNN, we try to minimize the error between the desired output and the actual output of the network, given an input. In other words, given an input vector x and a known desired output vector y, we try to minimize the "loss" with respect to the FFNN parameters θ:

$$L(\theta : x, y) \tag{2.5}$$

In this case the parameters θ correspond to the weight matrices W^i and bias vectors b^i from equations 2.1. In this chapter we specify the loss function for a single training sample. In reality the losses are usually aggregated over a mini-batch. In this report, we wish to classify future price movements as either going up or down. This is a binary classification problem, and a suitable loss function is "log loss":

$$L(\theta; x, y) = y \log(y_p) + (1 - y) \log(1 - y_p) \tag{2.6}$$

Here y_p corresponds to the network prediction as in Eq. 2.1, and therefore is a function of x. In this case, we are assuming that y_p is the probability of the price going up and y is the truth (1 if it does go up, and 0 otherwise). The sigmoid function should be used as activation function in the output layer. A similar loss function that we use is "scaled log loss". The function looks identical to "log loss", but during the weight updates contributions from different training samples are scaled by their absolute value. This has the effect of prioritizing larger movements over smaller movements in price.

We have also used the "mean squared error" and "mean absolute error" loss functions:

$$L(\theta; x, y) = (y - y_p)^2 \tag{2.7}$$

$$L(\theta; x, y) = |y - y_p| \tag{2.8}$$

These loss functions attempt to predict the price change directly. Therefore, y is the true percentage price change, and y_p is the prediction. No activation function (linear activation) was used in the final layer when using these loss functions. Another loss function that has been used in this report is "mean absolute profit loss" (mapl):

$$L(\theta; x, y) = |y| - y_p y \tag{2.9}$$

Here, we assume that the output layer activation function is tanh, and therefore y_p lies in range $[-1, 1]$, where negative values indicate a price drop, and positive values indicate a rise. The value of y is the true percentage price change. The maximum profit for a training sample is then $|y|$. Therefore, if y_p has the same sign as y, the term $y_p y$ will be positive, and be subtracted from the maximum possible profit $|y|$. This loss function then measures how much "potential profit" is lost. This is similar to the "mean absolute error" loss function, except it is scaled by the absolute value of the true percentage price change. This loss function has also been tested in the "squared" version "mean squared profit loss"(mspl):

$$L(\theta; x, y) = (|y| - y_p y)^2 \tag{2.10}$$

2.1.3 Regularization

Neural networks can have millions of neurons and can learn very complex functions. As a result, over fitting is a concern. Regularization methods put constraints on the model to prevent it learning too complex functions (over fitting). In this subsection we describe the methods of regularization that are used with FFNNs in this report. We use three main methods for regularization: L1, L2, and dropout. L1 regularization adds a penalty term to the loss function which penalizes the weights of the network:

$$C_{L1}(\theta) = \frac{\lambda_{L1}}{n} \sum_{n} |W_{ij}| \tag{2.11}$$

Here the sum is over all n elements of the weight matrices in each layer the regularization is applied to. The term λ_{L1} regulates how strong this

regularization should be. L1 regularization puts a restriction on the size of the weights of the model. The derivative of $C_{L1}(\theta)$ with respect to any single weight W_{ij}, is $sign(W_{ij})$. Therefore the magnitude of the weights are decreased by a constant amount with this type of regularization. This means that some weights will approach zero, while others will be non zero, making the weight matrices more sparse.

L2 regularization is very similar to L1:

$$C_{L2}(\theta) = \frac{\lambda_{L2}}{n} \sum_{n} W_{ij}^2 \tag{2.12}$$

In contrast to L1 regularization, the derivative with respect to any weight W_{ij} is proportional to the weight itself. This means that lower weights are lowered by smaller amounts than higher weights. The result is an overall penalty on the size of the allowed weights.

These penalties can simply be added to the loss function:

$$J(\theta; x, y) = L(\theta; x, y) + C, \tag{2.13}$$

where $J(\theta : x, y)$ is the complete loss function.

The third type of regularization used in this report is dropout. Dropout randomly ignores neurons during training with a certain probability. This means that for each training batch a different sub-network is being trained. When the network is finally being used, the outputs of each neuron are scaled, depending on the dropout probability. This has the effect of acting like an ensemble, each sub-network in the ensemble contributes to the final prediction.

2.1.4 Batch Normalization

Batch Normalization is used to speed up the training time for neural networks [15]. Batch Normalization works by normalizing the outputs of each layer (before activation is applied) to have zero mean and unit variance. This places the outputs in the "interesting range" of the activation functions, since most activation functions are centered around zero. Batch Normalization normalizes the outputs of each layer using the mean and standard deviation of each batch. Let m be the batch size and x be the output from a hidden layer, then the algorithm is as follows:

$$\begin{aligned} \mu &= \frac{1}{m}\sum_{i=1}^{m} x_i \\ \sigma^2 &= \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2 \\ \hat{x} &= \frac{x-\mu}{\sqrt{\sigma^2 + \eta}} \\ BN(x;\gamma,\beta) &= \gamma\hat{x} + \beta \end{aligned} \tag{2.14}$$

Here γ and β are learn-able parameters and η is a small constant for numerical stability. The purpose of these parameters is for the network to have the option to modify (or undo) the normalization. During training, the mean (μ) and variance (σ^2) of each output are aggregated in an exponentially decaying moving average. These averages are then used outside of training.

2.1.5 Training

The goal of training, is to find a set of model parameters θ that minimize a loss function (including regularization penalties) over some known training set (X_{train}, Y_{train}), where X_{train} is a set of input vectors, and Y_{train} is a set of output vectors. There are multiple methods to solve this problem, most of which include some kind of gradient descent. The most basic algorithm calculates the gradient of the complete loss function (J) with respect to each model parameter (λ_i):

$$\partial J/\partial\theta_i \tag{2.15}$$

Because the whole network is composed of differentiable functions, these derivatives can be calculated using repeated use of the chain rule for differentiation. The weight θ_i is then updated with:

$$\theta_i \leftarrow \theta_i - \eta\frac{\partial J}{\partial\theta_i} \tag{2.16}$$

Here η is a parameter known as the learning rate. This update can be done for each training sample individually, but it is most commonly done in mini-batches. A mini-batch is a random subset of the training data. The loss is calculated for the entire mini-batch, and a single update is done. This has the advantage of being more computationally efficient, and also more "stable".

In this report, we make use of an algorithm called Adam [16] for the training process. The algorithm is presented in Alg 17.

Algorithm 1: Adam

1 α : step size;
2 $\beta_1, \beta_2 \in [0, 1)$;
3 $J(\theta; y, y_p)$: loss function ;
4 θ_0 : initial parameters;
5 $m_0 \leftarrow 0$ (Initialize running gradient average);
6 $v_0 \leftarrow 0$ (Initialize running gradient^2 average);
7 $t \leftarrow 0$;
8 $\epsilon \leftarrow 10^{-8}$ (constant for preventing division by zero);
9 **while** *training* **do**
10 $\quad t \leftarrow t + 1$;
11 $\quad g_t \leftarrow \nabla_\theta J(\theta_{t-1}; y^t, y_p^t)$ (calculate gradients using current minibatch);
12 $\quad m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ (exponential moving average of gradients);
13 $\quad v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ (exponential moving average of squared gradients);
14 $\quad \hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (correct for zero initialization);
15 $\quad \hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (correct for zero initialization);
16 $\quad \theta_t \leftarrow \theta_{t-1} - \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$;
17 **end**

The actual weight update can be seen in line 16. Note the similarity to Eq. 2.16. Instead of updating with the current gradient, Adam uses a running exponential moving average of the gradients m_t. The learning rate from Eq. 2.16 is replaced with the term $\alpha / (\sqrt{\hat{v}_t} + \epsilon)$. This means the learning rate is adjusted based on the standard deviation of the gradients. This is good, because we want to make bigger steps when the gradients are not changing much (far from optimum), and smaller steps when the gradients vary a lot (close to minimum).

2.2 Long Short Term Memory (LSTM)

In this section we describe the basic elements of a LSTM cell (Fig. 2.2). A LSTM neural network attempts to capture long term relations between inputs in sequence. This is achieved by the cell state, labeled C_t in Fig 2.2. At each time step, the cell state is only modified twice: by the

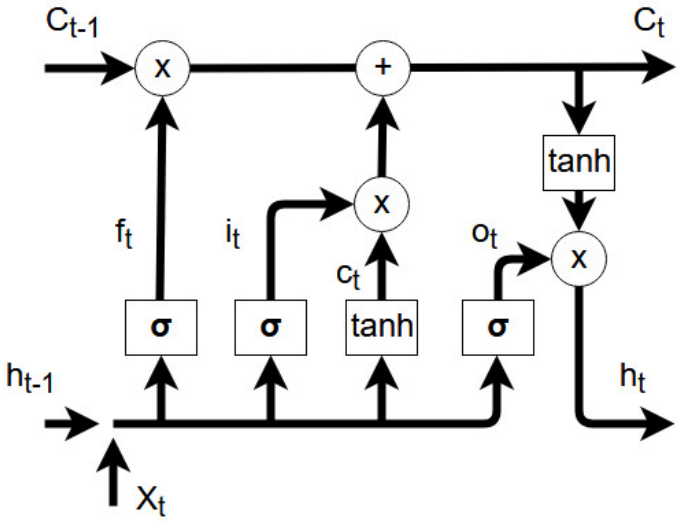


Figure 2.2: A LSTM cell

"forget" path and the "input" path. Each modification is only a point wise multiplication or addition, meaning that information can theoretically pass unchanged between time steps. This is in contrast to classic Recurrent Neural Networks (RNNs) where the states are multiplied by a weight matrix and fed through a non linear activation function between each time step.

Starting at the bottom left of Fig 2.2, we see the input vector X_t at time t. It is concatenated with the output of the previous time step h_{t-1} and sent to different elements in the cell. The first from the left, is the "forget gate". It consists of a weight matrix W_f, a bias vector b_f and a sigmoid activation function σ. The output f_t is:

$$f_t = \sigma(W_f Z_t + b_f) \tag{2.17}$$

where

$$Z_t = [h_{t-1}, X_t] \tag{2.18}$$

is the concatenation of X_t and h_{t-1}. Because we apply the σ activation function in 2.17, the resulting vector f_t will have values between zero and one. When we point-wise multiply f_t with the previous cell state C_{t-1}, it will have an effect of reducing the elements of C_{t-1} whose corresponding elements of f_t are close to zero. In other words, some elements of C_{t-1} are suppressed, or "forgotten".

The next element is called the input gate, with output i_t. It behaves exactly the same as the forget gate, except it is not point wise multiplied with the cell state. The output i_t is instead point wise multiplied with the output of the next element in the LSTM cell: the candidate values. This gate also has a weight matrix W_i and a bias term b_i associated with it.

The candidate values c_t can be seen as possible updates to the current cell state. They are calculated with:

$$c_t = \tanh(W_c Z_t + b_c) \tag{2.19}$$

The equation is almost identical to the previous two elements, with the exception of using a $\tanh$ activation function instead of σ. This means that the output values c_t are scaled to be between -1 and 1. This vector contains possible updates that can be done to each element of the cell state C_{t-1}. However, only some of these updates are fully passed on to the cell state. This is done by point wise multiplying with the output of the previous element, i_t, which is scaled 0 to 1. Conceptually, the job of c_t is to propose updates to the cell state, and the job of i_t is to filter out which ones are important enough. The result is point wise added to the state. The full update to the cell state is then:

$$C_t = f_t \odot C_{t-1} + i_t \odot c_t \tag{2.20}$$

Here $\odot$ means point wise multiplication. Next is the output gate, with a similar function to f_t and i_t. The output, o_t if used to filter what parts of the memory are to be outputted out of the cell. This element also has a weight matrix W_o and bias vector b_o associated with it. The output o_t is point wise multiplied by a scaled version of the updated cell state C_t. The scaling is done by taking the $\tanh$ of each element. The cell output h_t is then fed into the next layer, and also into the next time step.

$$h_t = o_t \odot \tanh(C_t) \tag{2.21}$$

When fed into the next layer, an activation function is applied. For the final output, a fully connected layer is connected to the final LSTM layer, such that the input to this layer is $f(h_t)$. This fully connected layer then has an output size and an activation function to produce the desired output format.

The size of C_t (and consequently the output size h_t) controls how powerful the cell is, with a larger value giving the LSTM cell more "memory slots".

Everything that was previously discussed about FFNNs such as loss functions and regularization can be applied to LSTMs.

2.3 Q-Network

Reinforcement learning is often applied to learn sequences of actions, in this case changes in positions, when the outcome (reward) of these actions are delayed and it is hard to assign a reward to a particular action. For this thesis we have adopted Q-learning and used a neural network to learn a so called Q-network based on [12]. The goal of the Q-network is to estimate the expected future reward of each possible action, given a world state. In our case, a world state consists of a feature vector calculated from the time series, plus one value that tells us the last action taken. The last action is important, because changing positions from short to long or long to short incurs a penalty based on the bid-ask spread and is taken into account when rewarding the network in the learning process. Our Q-Network is a FFNN so everything previously discussed about FFNNs also applied here. We only consider two actions: going short and going long. The output type is a floating point value with no bounds. Therefore the Q-network has two nodes in the output layer, and there is no activation function used (linear).

We want the Q-network to predict the expected future reward. The output of the network is:

$$Q(x_t) = \begin{bmatrix} q_s \\ q_l \end{bmatrix} \tag{2.22}$$

Where q_s is the expected future reward (or quality) of taking action "short" given state x_t, and q_l is the same but for action "long". We wish to maximize future reward, so the action taken at time t is then:

$$a_t = argmax(Q(x_t)) \tag{2.23}$$

The feature vector for time $t + 1$ is then modified to reflect that the last action taken was a_t. This is encapsulated in a "transition function"

which returns a reward and a next state, given a previous state and an action:

$$(r_{t+1}, x_{t+1}) \leftarrow T(x_t, a_t) \tag{2.24}$$

The reward is the returns for holding the position a_t for that day, minus a transaction cost if $a_t \neq a_{t-1}$. The next state is simply the feature vector for the next time step, and a_t.

2.3.1 Training

The network is trained in the same way as it is presented in [12]. The network steps through the data, one time step at a time and makes a prediction according to Eq 2.23. There is a small probability α to instead take a random action (exploration). Let x_t be the currently considered time step. Then:

$$a_t = \begin{cases} random([0,1]) & \text{with probability } \alpha \\ argmax(Q(x_t)) & \text{else} \end{cases} \tag{2.25}$$

Then we get a reward (r_t) and next state (x_{t+1}) according to Eq. 2.24. We then store the tuple (x_t, a_t, r_t, x_{t+1}) in a "experience replay" buffer. The reason for this buffer is to de-correlate successive training samples. This buffer has a size limit and when it is filled the oldest entry is deleted to make room for a new entry. Assume the buffer is full. A batch of tuples are randomly sampled from the buffer. The batch size is equivalent to the batch size in FFNNs. For every tuple in the batch, we calculate:

$$q = r_t + \gamma \max_a Q(x_{t+1}) \tag{2.26}$$

This value is the immediate reward (r_t) from taking action a_t at state x_t, plus a discounted value of the maximum expected reward from the next state ($\max_a Q(x_{t+1})$). The discount factor γ is a hyper parameter. We then form a supervised training set, where the training features are the x_t, and the targets are:

$$y_i = \begin{cases} Q(x_t)_i & i \neq a_t \\ q & y = a_t \end{cases} \tag{2.27}$$

Note that y is a vector of size 2. The element that corresponds to the action taken is set to q and the other element is left at the current output of the network, given the state x_t. This means that when training on this sample (x_t, y), the network will only update its policy for the action a_t, because the target and output are already the same for the other element. This is what we want, because we want the network to learn from the actions it takes. If it has performed an action at time t, we want it to learn the consequences of that action, and not the action it didn't take.

2.4 XGBoost

XGBoost [4] is an implementation of the Gradient Boosting algorithm. Like other boosting methods, XGBoost uses an ensemble of weak learners to make predictions. Specifically XGBoost uses an ensemble of decision trees, which makes the algorithm very similar to a random forest. The individual trees are trained in sequence (unlike random forests) where each successive tree is trained on the "pseudo-residuals" of the previous model. Let $L(y, y_p)$ be a loss function where y is the truth value and y_p is the predicted value. An overview of the training algorithm can be seen in Alg 2.

Algorithm 2: Gradient Boosting with Decision Trees

1 x, y : training dataset;
2 $\alpha \in [0, 1]$: shrinkage;
3 M : number of iterations;
4 $L(y, y_p)$: loss function ;
5 f : Weak learner (decision tree);
6 $F_0 \leftarrow \arg\min_a \sum_{i=1}^{n} L(y_i, a)$ (initialize model);
7 $m \leftarrow 1$;
8 **while** $m \leq M$ **do**
9 $\quad r_i \leftarrow -\left(\frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}\right) \forall i$ (partial residuals);
10 $\quad f_m \leftarrow$ fit weak learner to dataset (x, r);
11 $\quad \gamma_m \leftarrow \arg\min_{\gamma_m} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \gamma_m f_m(x_i))$;
12 $\quad F_m \leftarrow F_{m-1} + \alpha \gamma_m f_m$;
13 $\quad m \leftarrow m + 1$;
14 **end**
15 **return** F_M;

Line 9 of the algorithm calculates the training targets of the next tree as the gradients of the errors of the full model at the previous iteration. Line 10 performs the fit using the training algorithm associated with the weak learner f. In our case, we use decision trees. The splits of the decision tree are done using a combination of the loss function, and a "structure score" which penalizes complex trees (regularization term). The structure score penalizes tree depth and leaf scores. Leaf scores are penalized with L2 regularization, with a L2 weight parameter λ_{L2}. The tree depth is not penalized in our model, but is limited to a maximum tree depth.

The loss function used with XGBoost in this report is log loss, because we are doing binary classification. Line 11 computes a optimal step size γ_m to take in the direction of the gradients from line 9. In line 12 we update the current model with the newly learned tree. The new tree, which was trained to predict gradients of the error, is multiplied by γ_m and the shrinkage parameter α before being added to the model. The shrinkage parameter is a regularization term which prevents the newly created tree from "explaining too much" by reducing the step size γ_m.

Another regularization parameter we consider is the "column sub-

sampling rate". At each iteration of the training algorithm, we subsample the columns (features) of the training set. This way, each individual tree is trained on a potentially different set of features. This is very similar to a technique used in random forests.

The XGBoost model overfits very easily, due to the training process. Given enough iterations the model will fit the entire dataset. Therefore regularization is very important.

Chapter 3

Methodology

3.1 Software

All experiments were done using Python and the scikit-learn library [17]. All neural networks made use of Tensorflow [18], and the XGBoost [4] implementation of Boosted trees was used. All data management was done using the Pandas library [19].

3.2 Overview

Figure 3.1 shows the overall process. First the data is pre-processed, then split into validation and evaluation sets. To choose hyper-parameters for each model, model validation is performed on the validation set by predicting a single time series (XOver_TR). The best model is then trained on both the validation and evaluation sets. Results on the evaluation set are recorded. Testing on the evaluation set is repeated 50 times independently.

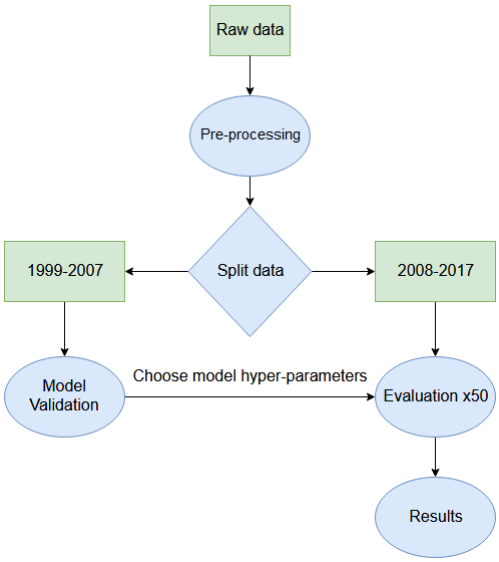


Figure 3.1: An overview of the validation and evaluation process. Validation is performed on years 1999-2007 on the XOver_TR time series, and evaluation is performed on 2008-2017.

3.3 The Data

Data from 1999-01-01 to 2017-12-31 was used in this report. The data includes daily close prices for several time series, shown in Table 3.1. In this report we want to predict the bond indexes EUR_HY_TR, EU_Corp_TR, YS_HY_TR, US_Corp_TR and XOver_TR.

Time Series Code	Description
EURUSD	EUR USD exchange rate
EUR_HY_TR	Bloomberg Barclays Pan-European High Yield Total Return Index
EU_Corp_TR	Bloomberg Barclays Euro Aggregate Corporate Total Return Index
EuroStoxx50	Eurostoxx 50 Index
Gold	Spot price of gold
MSCI_World	MSCI World Index
OMX	OMX 30 Stockholm Index
Oil	Oil Futures
SP500	S&P 500 Index
US_Corp_TR	Bloomberg Barclays US Credit Total Return Index
US_HY_TR	Bloomberg Barclays US Corporate High Yield Total Return Index
US_Treasury	Bloomberg Barclays US Treasury Total Return
VIX	US S&P 500 implied Volatility index
VSTOXX	Eurostoxx implied Volatility index
XOver_TR	Long position in XOver HY CDS Total Return Index

Table 3.1: Available data for this study

3.3.1 Data pre-processing

Missing values were filled in by the last non-missing value. Features were calculated for each time step (day) for each time series. The features that were calculated are covered later in the report. The features of time series XOver_TR were padded with values from EUR_HY_TR from 1999-01-01 to 2004-06-22. This is because data is available for XOver_TR from 2004-06-23 onwards. XOver_TR is a crossover bond index which contains bonds from both investment grade and high yield categories, and is therefore somewhat correlated with EUR_HY_TR. Before training, the features are normalized to have zero mean and unit variance. The features are also PCA transformed and only the first m components are used for training. m is chosen such that the first m components explain 98% of the variance. The value 98% was chosen so that roughly 100 components would be selected. Note that

both the scaling parameters and the PCA transformation were calculated on the training sets, and then applied to the test sets using the parameters from the training sets.

When training the LSTM model, one further step of pre processing is performed. All features are split up into smaller time series of length 20. Let X be the complete data matrix with dimensions $n \times k$, where n is the number of time steps and k is the number of features. The data is transformed into $n-20$ separate time series of size 20. The final data matrix $\hat{X}$ is then $(n-20) \times 20 \times k$ where $\hat{X}_i = X_{i:(i+20)}$. The reason for this is to allow for random shuffling of the training data during training.

3.4 Functions

In this section, we introduce some functions and notation used for defining the features that we extract.

Exponential Moving Average (EMA)

The exponential moving average is a weighted average of a time series X, where the weights are exponentially decaying. This means that the value of the EMA at time t is influenced more by values of X closely before X_t than values further away. The EMA has one parameter α which governs the rate of decay of the weights. The EMA is useful as a kind of smoothing filter which gives more weight to more recent values. The recursive formula for the EMA is:

$$EMA(X, \alpha)_1 = X_1 \tag{3.1}$$

$$EMA(X, \alpha)_t = \alpha X_t + (1-\alpha) EMA(X, \alpha)_{t-1} \tag{3.2}$$

The parameter α is sometimes given in terms of m:

$$\alpha = \frac{2}{1+m} \tag{3.3}$$

This is then the "m-day EMA". From now on in this report, the Exponential Moving Average will be specified using m: $EMA(X, m)$.

Mean

The mean of a vector X is given by:

$$mean(X) = \frac{\sum_{i=1}^{t} X_i}{t} \tag{3.4}$$

Standard Deviation

The standard deviation of a vector X is given by:

$$std(X) = \sqrt{\frac{\sum_{i=1}^{t}(X_i - mean(X))^2}{t-1}} \tag{3.5}$$

Exponential Moving Standard Deviation

The m-day exponential moving standard deviation of a vector X is given by:

$$\begin{aligned} EMSTD(X,m)_t &= \sqrt{\alpha(X_t - EMA(X,m)_t)^2 + (1-\alpha)EMSTD(X,m)_{t-1}^2} \\ \alpha &= \frac{2}{1+m} \end{aligned} \tag{3.6}$$

Minimum and Maximum

The minimum and maximum values of a vector X are given by $min(X)$ and $max(X)$, respectively.

3.5 Features

This section defines some functions and features that are used in later sections. Let C be the vector that holds the close price time series. After all features are computed, they are normalized to have mean 0 and standard deviation 1 (using the mean and standard deviation of the training set). Each feature is calculated for each ticker (time series). The vector of all features is then used as input into all forecasting models.

N-day return

This feature measures how much the price has changed since n days ago. The change is expressed as a percentage.

$$return(C, n)_t = \frac{C_t - C_{t-n}}{C_{t-n}} \tag{3.7}$$

The n parameter has a sort of smoothing function. The larger n is, the less the n day return is affected by small perturbations. This is a common feature that has been used by several papers in the literature, and has also been selected by the feature selection process of Longmore (2016) [8] [9].

Mean of returns

This feature is the n day EMA of 1-day returns:

$$mean\_of\_returns(C, n) = EMA(return(C, 1), n) \tag{3.8}$$

This feature is an estimate of the average price movement over the last n days, weighted towards the more recent days.

Standard deviation of returns

This feature is the exponential moving standard deviation of the last n days of 1-day returns:

$$std\_of\_returns(C, n) = EMSTD(return(C, 1), n) \tag{3.9}$$

This feature is an estimate of the volatility over the last n days, weighted towards the more recent days.

Trend

The m day trend ($trend(C, m)$) is the slope of a line fitted through the last m points. The fit is done using least squares. First, the last m points are extracted:

$$Y = C_{t-m+1:t} \tag{3.10}$$

The vector Y is normalized by the first value:

$$Y = Y/Y_0 \tag{3.11}$$

Let X be the vector $[0, 1, 2, ...m - 1]^T$. Then a line is fit through (X, Y). The slope of the line is then $trend(C, m)$. A positive value of the m day trend is that the price is rising, while a negative value means it is dropping. The parameter m decides on the scale of this trend. A larger m tries to capture large scale price movements, while lower m tries to capture smaller scale movements. A similar feature was selected by Longmore (2016) [8] [9].

Momentum

The m-day momentum is given by:

$$momentum(C, m) = return(C, m)/std\_of\_returns(C, m) \tag{3.12}$$

This feature can be seen as a normalized version of n-day return, where we normalize by the standard deviation of returns over the same period. This has the effect of penalizing time periods with higher volatility. A similar feature performed well in [8].

EMA difference

The n-day EMA difference (EMAD) is given by:

$$EMAD(C, n) = (C - EMA(C, n))/C \tag{3.13}$$

This is the current price minus the n day EMA, scaled by the current price. This feature shows how far the current price is from the trend.

Stochastic price oscillator

The stochastic price oscillator (SPO) measures the current price in relation to the minimum and maximum price in a n-day window. The equation is given by:

$$SPO(C, n)_i = \frac{C_i - min(C_{(i-n):i})}{max(C_{(i-n):i}) - min(C_{(i-n):i})} \tag{3.14}$$

The SPO is bounded by 0 and 1. When the SPO is 0 that means the price is the lowest is has been in the past n days, and when it is 1 it means the price is the highest it has been in the past n days.

Absolute price oscillator

The Absolute Price Oscillator (APO) is given by the difference between a shorter and a longer period of EMA of the price, normalized by the current price:

$$APO(C, m, n) = (EMA(C, m) - EMA(C, n))/C \tag{3.15}$$

where $m < n$.

When the APO is negative, this indicates a downward trend, and when positive, an upward trend. A similar feature was selected by Longmore (2016) [8] [9].

3.5.1 Used features

The table below lists all features that were used in some experiment.

Feature
$return(C, 1)$
$return(C, 5)$
$return(C, 10)$
$std\_of\_returns(C, 10)$
$std\_of\_returns(C, 20)$
$std\_of\_returns(C, 100)$
$trend(C, 10)$
$trend(C, 20)$
$trend(C, 100)$
$momentum(C, 10)$
$momentum(C, 20)$
$momentum(C, 100)$
$EMAD(C, 10)$
$EMAD(C, 20)$
$EMAD(C, 100)$
$EMAD(C, 200)$
$SPO(C, 10)$
$SPO(C, 20)$
$SPO(C, 100)$
$APO(C, 10, 30)$
$APO(C, 30, 100)$
$APO(C, 100, 200)$

Table 3.2: Features with parameters that were used in this study.

3.6 Forecasting

Each model considered in this report attempts to forecast the price movement for the next day. Let X_t be a vector of all features calculated on all tickers at time t. Then, the goal of the forecasting models is to estimate the sign of the N-day return at time $t + n$. So, the goal is to forecast

$$s_t^n = sign(return(C, n)_{t+n}) \tag{3.16}$$

This means that the model is forecasting in which direction the price will change from today to n days from now. With $n = 1$, we predict the price change from t to $t + 1$.

In practise, when we want to make a prediction for the next day, we still do not have the close prices for the current day. There is also an issue that some close prices are made available earlier than others due to time zones. To eliminate these issues, the features are lagged one day:

$$X_t^{lagged} = X_{t-1} \tag{3.17}$$

This time lag is done for both training and testing of the models. Henceforth, when we mention features at time t, we mean the features that are used for prediction at time t, in other words the features that are calculated using data from time $t-1$. Although we lose potential information by doing this, we eliminate any possibility of the features being contaminated with future information.

In this report, we do forecasting as follows. The models are trained to estimate s_t^n, with X_t as features. The actual output of the model depends on the loss function. Depending on the loss function used, and model architecture, the model may output a direct classification ($\{-1, 1\}$), a regression ($\Re$), or a confidence value ($[0, 1]$). In any case, the output is converted into a value $\tilde{s}_t$ in the range $[-1, 1]$, where positive values indicate the models confidence prices rising and negative values indicate confidence in price dropping. The prediction for the next day is then a combination of the predictions of the last n days. This combination is done either as an average, or as a vote. When voting, each prediction is weighted the same, regardless of model confidence.

For example assume we are at $t = 2$ and we wish to make a prediction for $t = 3$. Let us also assume that $n = 3$. We than make a prediction at $t = 0, 1, 2$, and take the average prediction for $t = 3$. This is illustrated in Table 3.3.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6
$\tilde{s}_0$	-	$+1$	$+1$	$+1$	-	-	-
$\tilde{s}_1$	-	-	-0.5	-0.5	-0.5	-	-
$\tilde{s}_2$	-	-	-	$+1$	$+1$	$+1$	-
vote				$1/3$			
mean				$1/2$			

Table 3.3: Illustration of forecasting process with $n = 3$. At $t = 0$ the model predicts the price will rise the next 3 days. At $t = 1$ that it will fall, and at $t = 2$ that it will rise. When voting, the total prediction for $t = 3$ is then the average of the sign of the last $n = 3$ predictions: $(1 - 1 + 1)/3 = 1/3$. When taking the mean, it is $(1 - 0.5 + 1)/3 = 1/2$

The advantage of setting $n > 1$ is that the predictions will tend to have larger streaks of long and short positions. This is good because every time we change position money is lost to transaction fees. Another advantage of setting a larger n is that it acts as a sort of smoothing filter for the model. The models will attempt to predict the larger scale patterns, and not the day-to-day noise. It also functions as an average over several models. If we were to predict with n days ahead and only make a prediction every n days, then the model depends on what day we begin. We can start on day i and then make another prediction at $i + n$, etc. We can also start on day $i + 1$ and the next prediction would be $i + 1 + n$ and so on... There are n possible starting days (offsets), and when we perform the voting as illustrated in Table 3.3, we get an average of n models. This is illustrated in Figure 3.2. The red line is the result of voting among the 11 models as in Table 3.3.

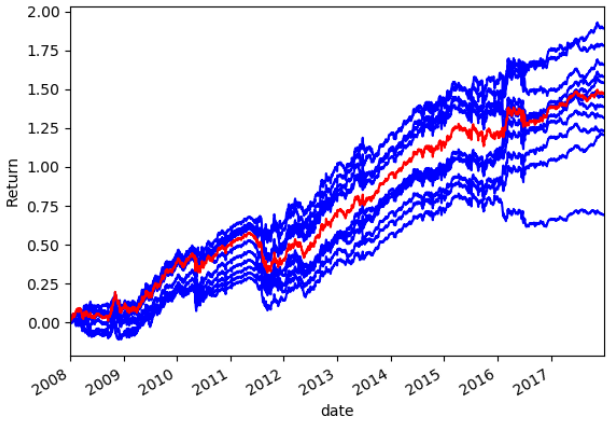


Figure 3.2: Results of testing on XOver_TR from 2008-2017 with $n = 11$. The plot shows returns over time. There are 11 plots corresponding to returns of 11 models each offset by one day (blue). The average by voting of the 11 models is show in red.

3.7 Evaluation

3.7.1 Performance metrics

In this report, we use Sharpe Ratio when evaluating the performance of models. The annualized Sharpe ratio is defined as:

$$Sharpe(r) = \sqrt{252} \cdot \frac{mean(r - r_f)}{std(r)} \tag{3.18}$$

When applied to the daily asset returns (r), it measures profitability (mean) which is penalized by volatility (std). The $\sqrt{252}$ term appears to "annualize" the ratio; there are approximately 252 trading days in a year. We omit the risk free rate r_f from the formula as it is negligible.

We also apply a penalty when the model decides to change position (short to long or long to short) of 0.03%. This is approximately the real value for the XOver_TR index at the time this work was done.

3.8 Validation and Testing

Model validation was done by predicting the XOver_TR index during the years $1999 - 2007$. Training and testing was performed in yearly increments by training from year 1999 to year y_i, and testing on year y_{i+1}, as seen in Table 3.4. To have enough training data for model validation, testing was performed only from year 2008.

	...	2008	2009	2010	2011	...
iteration 1	train	test	-	-	-	-
iteration 2	train	train	test	-	-	-
iteration 3	train	train	train	test	-	-
iteration 4	train	train	train	train	test	-
iteration n	...					

Table 3.4: Validation/testing strategy

The above technique was applied to the credit index time series, and the test results were aggregated over all years to calculate an overall Sharpe ratio which was used to compare to the Sharpe ratio of a buy-and-hold strategy.

For model validation, this process was repeated several times using a randomized search over the hyper-parameter space. Finally, each hyper parameter was selected as the one that had the highest average sharpe ratio during validation.

3.9 Neural network architecture

We have constrained our network architecture to have two hidden layers. We do manual feature extraction and we do not have that many data points to justify a deeper architecture. The architecture can be seen in Table 3.5.

Layers	Notes / Hyper Parameters
Input Layer	All features presented in Table 3.2 calculated on all series presented in Table 3.1, z-normalized
Fully Connected Layer 1	Layer size (h), L1 weight (λ_{L1}), L2 weight (λ_{L2}), dropout probability (p_d)
Batch Normalization	-
Activation	Relu
Fully Connected Layer 2	Layer size (h), L1 weight (λ_{L1}), L2 weight (λ_{L2}), dropout probability (p_d)
Batch Normalization	-
Activation	Relu
Output layer	fully connected layer with size 1
Activation	dependant on loss function

Table 3.5: Architecture of the FFNN that was used. The hyper parameters are chosen using a random search on the XOver_TR data during years 1999 - 2007.

To choose the hyper parameters, a random search was performed as described in section 3.8. The parameters to the Adam optimizer were left at their default values ($\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$). The number of training epochs was found by analysing the training and validation losses on the validation set (1999-2007). The reason for not using early stopping is that the result would be heavily biased on the validation set. When the validation set included a year where the price just rises, then the predictions on the test set would be biased towards going long. The same effect was observed when the validation set included a year where the price dropped. Therefore the number of epochs was set to a constant number.

We use the same architecture for LSTM and the Q-Network as for FFNN with the exception that the Q-Network has an output layer of size 2 and linear activation.

Chapter 4

Results & Discussion

4.1 Model Validation

4.1.1 FFNN

The results of model validation of FFNN are presented here. The best parameters found experimentally are presented in Table 4.1.

Parameter	Value
Layer size (h)	16
L1 weight (λ_{L1})	0.0
L2 weight (λ_{L2})	0.01
Dropout (p_d)	0.5
Loss	logloss
n	11
Combine mode	vote
Training epochs	50

Table 4.1: Results of FFNN validation on years 1999 - 2007. Random parameter search was done using 250 models.

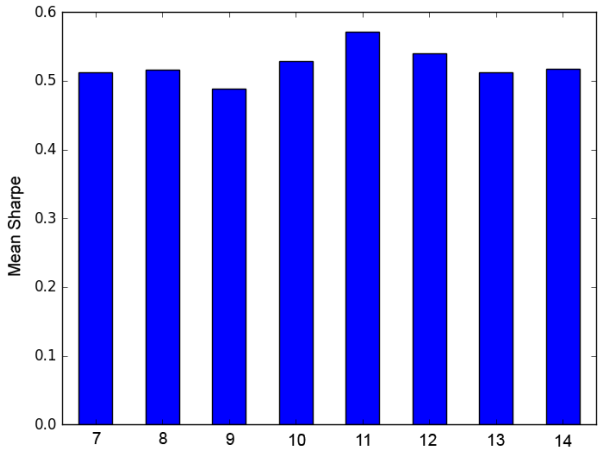


Figure 4.1: Validation results of the n parameter. The FFNN model is quite robust to different settings for n. Average Sharpe Ratio on y axis.

The validation result for the n parameter (n days ahead prediction) was used for all other models in this report as well. See Fig 4.1 for the validation results for the n parameter. The setting $n = 11$ seems to be marginally better than the alternatives. However, there have only been 250 random sets of parameters tested in the random search due to time constraints. The FFNN has shown to be robust to all parameter choices, with the exception of loss function and dropout, presented in Fig 4.2 and Fig 4.3.

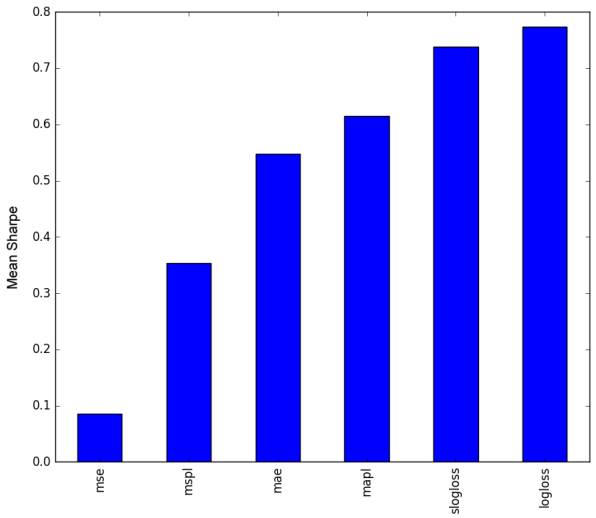


Figure 4.2: Validation results of the loss function. The FFNN model works best with logloss. Average Sharpe Ratio on y axis

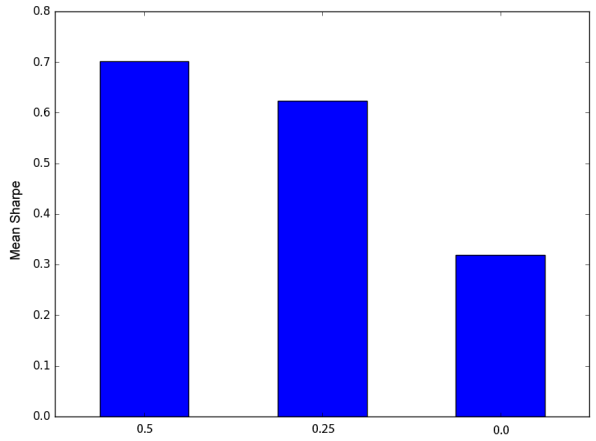


Figure 4.3: Validation results of the dropout rate. The FFNN model works best with 0.5 dropout. Average Sharpe Ratio on y axis

Loss function has a great effect on the results, with logloss being the best choice as seen in Fig 4.2. The motivation behind scaled logloss and mean absolute/square profit loss (mapl/mspl) is that it puts higher weights on training samples where the price changes by a larger amount. This makes sense, because we are not interested in

model accuracy, we are interested in profit. It is only logical that the loss function should reflect that. The validation however showed that the best loss function was regular logloss. One reason for this is perhaps that large movements in price are usually associated with unusual disturbances in the market. These disturbances are usually random and hard to predict, and by weighting them higher, we force the models to overfit.

Figure 4.4: Training loss during validation. Training from 1999 to 2006. Testing on 2007.

The number of training epochs was selected by observing the validation loss when training on the model validation set. An example of training and validation error curves can be seen in Fig 4.4. Note however that although the loss curves appear smooth, the Sharpe ratios behave almost randomly (Figure 4.5). This is due to a disconnect between the loss function and the evaluation function. The loss function is trying to maximize accuracy, while the evaluation function is measuring profit.

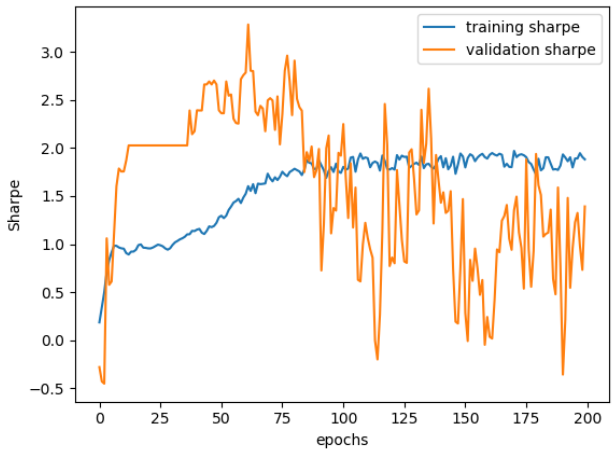


Figure 4.5: Sharpe ratios during validation. Training from 1999 to 2006. Testing on 2007.

4.1.2 LSTM

The same procedure as for FFNN was repeated for LSTM. The LSTM model proved to be much more difficult to train compared to FFNN. The training took significantly longer time and the results had very high variance. Therefore, not many random search iterations could be performed due to lack of time. The chosen parameters presented in Table 4.2 were the result of only 100 iterations of the random search. The value for n was not searched over, it was set to 11. As with FFNN, there was no clear difference in the choices of hyper parameters.

Parameter	Value
Layer size (h)	8
L1 weight (λ_{L1})	0.0
L2 weight (λ_{L2})	0.01
Dropout (p_d)	0.5
Loss	logloss
n	11
Combine mode	vote
Training epochs	30

Table 4.2: Results of FFNN validation on years 1999 - 2007. Random parameter search was done using 100 models.

4.1.3 XGBoost

Because XGBoost runs much faster, we could do many more iterations on the random parameter search. The search was run over 25000 iterations. The final parameter choices are summarized in Table 4.3.

Parameter	Value
n estimators (M)	40
Shrinkage (α)	0.001
L2 weight (λ_{L2})	0.0
Max tree depth	1
Column sub sampling	0.5
Loss	logloss
n	11

Table 4.3: Results of XGBoost validation on years 1999 - 2007. Random parameter search was done using 25000 models.

The model was very sensitive to the shrinkage, column sub-sampling and max depth parameters. The max depth parameter in particular was vary important, see Fig 4.6. For all parameter choices other than one, the model grossly over fits the training set, resulting in negative results. The XGBoost model over fits very easily due to the training process, so it makes sense that the regularization parameters are extremely important.

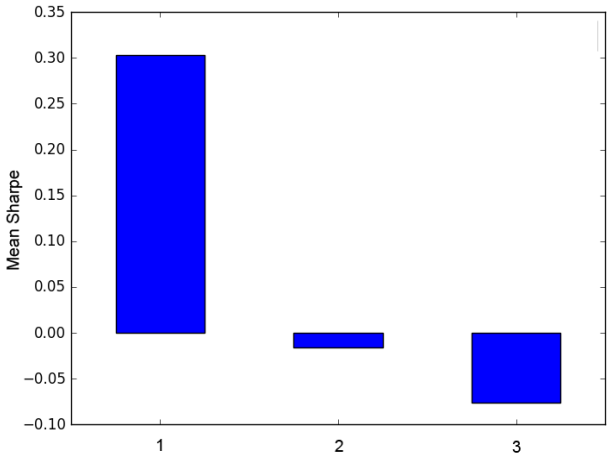


Figure 4.6: Validation results of the max tree depth parameter. The XGBoost model is very sensitive to different settings for this parameter. Average Sharpe Ratio on y axis.

4.2 Evaluation set results

Although the random parameter search for the FFNN did not run for very many iterations, the model was quite robust to most parameter settings. The n day ahead parameter was crucial to getting better results, as Table 4.4 shows. Increasing n from 1 to 11 improved the average Sharpe ratio, but also raised the standard deviation of the results. The randomness in the results is mainly due to the random weight initialization and random data shuffling during training.

Ticker	Strategy	n	Mean Sharpe	Std	Min	Max
XOver_TR	FFNN	11	1.09	0.14	0.64	1.35
	FFNN	1	0.83	0.05	0.71	0.96

Table 4.4: Effect of the "n days ahead" parameter n on XOver_TR. Statistics are over 50 independent tests from years 2008-2017.

Table 4.5 shows the effect of PCA on the FFNN and XGBoost models. It is interesting to note that PCA improves the results of FFNN in both mean and std, but worsens the results of XGBoost. This may be due to that it is very difficult for the FFNN to completely ignore some

features. To completely ignore a feature, the FFNN has to set several weights to zero during training. L1 regularization on the first layer of FFNN weight should force some weights to be zero, however the model validation process has selected the L1 weight to be zero.

Ticker	Strategy	PCA	Mean Sharpe	Std	Min	Max
XOver_TR	FFNN	Yes	1.09	0.14	0.64	1.35
	FFNN	No	0.99	0.21	0.32	1.46
	XGB	Yes	0.77	0.07	0.52	0.87
	XGB	No	1.21	0.09	1.05	1.38

Table 4.5: Effect of PCA pre-processing on XOver_TR. Statistics are over 50 independent tests from years 2008-2017.

XGBoost has no problems ignoring certain features by simply not picking them during the tree building process. Because the XGBoost model uses decision trees as weak learners, we can analyze which features mostly influence the results. The top five features used for predicting XOver_TR can be seen in Figure 4.7.

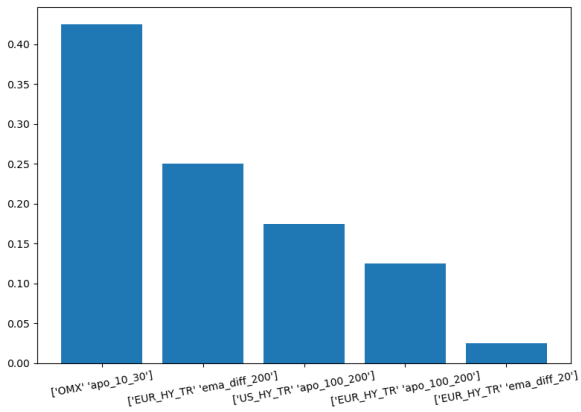


Figure 4.7: Top five feature importances of a random XGBoost model.

It is interesting to note that none of the features selected are from the XOver_TR time series. The most important feature is the $APO(10, 30)$ feature calculated on the OMX30 index. This is a relatively short-sighted feature since it is based on 10 and 30 day exponential moving averages, while the other features are based on 100 and 200 day

exponential moving averages. With only these features, the XGBoost model managed to outperform both the Buy&Hold strategy, and all other strategies implemented in this report.

Ticker	Strategy	Mean Sharpe	Std	Min	Max
XOver_TR	FFNN	1.09	0.14	0.64	1.35
	LSTM	0.68	0.24	0.28	1.19
	XGB	1.21	0.09	1.05	1.38
	Q	0.84	0	0.84	0.84
	Buy&Hold	0.84	0	0.84	0.84
EUR_HY_TR	FFNN	2.72	0.31	1.92	3.35
	XGB	2.88	0.17	2.43	3.12
	Buy&Hold	1.39	0	1.39	1.39
EU_Corp_TR	FFNN	1.59	0.15	1.28	1.95
	XGB	1.29	0.31	0.79	1.63
	Buy&Hold	1.68	0	1.68	1.68
US_HY_TR	FFNN	2.01	0.29	1.18	2.53
	XGB	1.91	0.11	1.62	2.09
	Buy&Hold	1.40	0	1.40	1.40
US_Corp_TR	FFNN	0.96	0.09	0.76	1.12
	XGB	1.07	0.02	1.01	1.11
	Buy&Hold	1.04	0	1.04	1.04

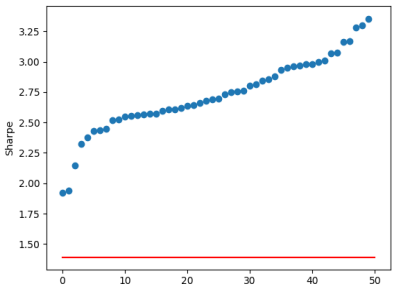
Table 4.6: Results of all models compared to Buy&Hold. Statistics are over 50 independent tests from years 2008-2017.

The final results of all models are summarized in Table 4.6. The XGBoost model managed to beat the Buy&Hold strategy on four out of five time series. XGBoost outperformed all other models, except for FFNN on the EU_Corp_TR and US_HY_TR time series. The FFNN model however had a much higher standard deviation in its results on the US_HY_TR time series and therefore XGBoost would be preferable to use in practise. On the EU_Corp_TR time series both FFNN and XGBoost failed to beat the Buy&Hold strategy, but FFNN outperformed XGBoost in both mean and standard deviation. Table 4.6 also tells us that both the FFNN and XGBoost models had troubles with the investment grade bond indexes (EU_Corp_TR and US_Corp_TR).

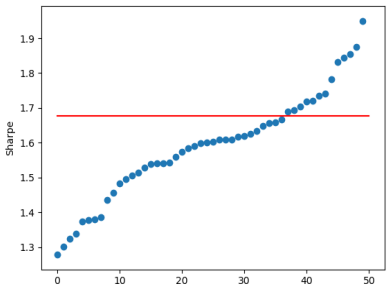
The LSTM model was very difficult to train, and both the mean and std of the Sharpe ratios were the worst among all tested models.

The theoretical advantage of the LSTM is that it is a recurrent neural network: it can take into account past values and find patterns in time. However, all of the features we have calculated already take into account historical patterns in some sense. Perhaps the LSTM model would be more suitable to be used with raw price as inputs and allow it to calculate its own features.

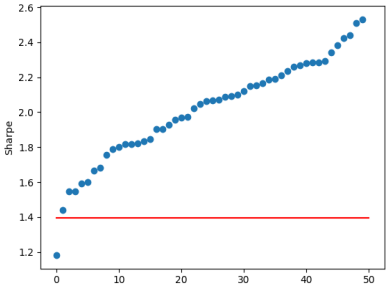
The Q-network only managed to learn to go long only and therefore got the exact same results as the Buy&Hold strategy. As previously mentioned, the Q-Network has not received as much attention as the other models in this report. On a positive note, it managed to beat the LSTM model.



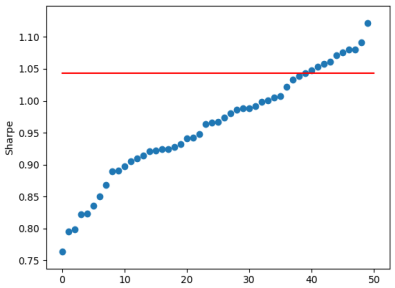
(a) EUR_HY_TR



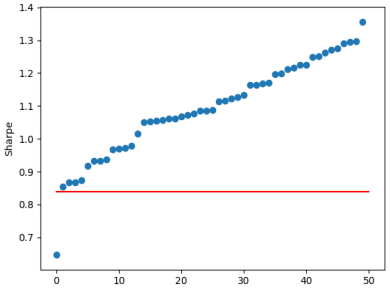
(b) EU_Corp_TR



(c) US_HY_TR

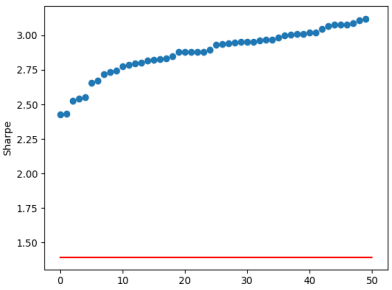


(d) US_Corp_TR

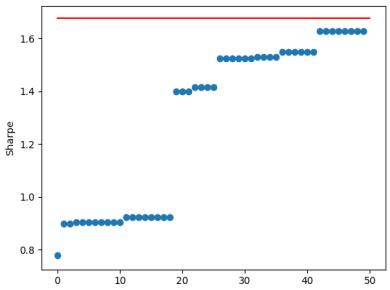


(e) XOver_TR

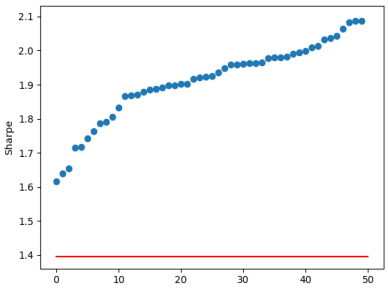
Figure 4.8: Results of FFNN on the test set. Model was trained on each target time series 50 times. Plots show performance of each individual model (blue) compared to the Buy&Hold strategy (red).



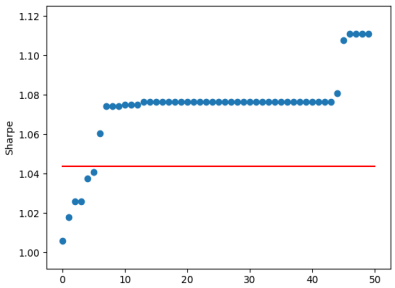
(a) EUR_HY_TR



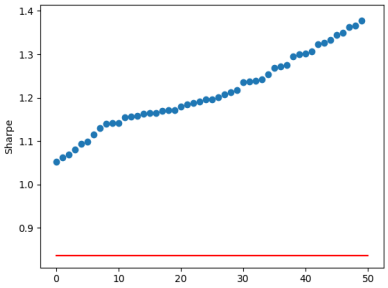
(b) EU_Corp_TR



(c) US_HY_TR



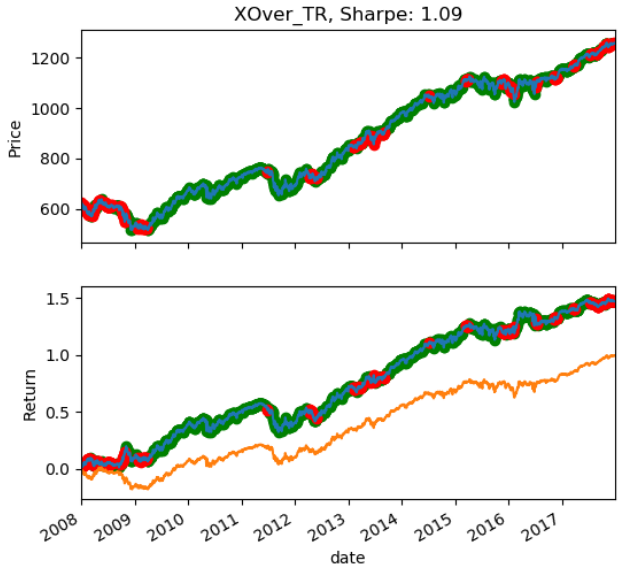
(d) US_Corp_TR



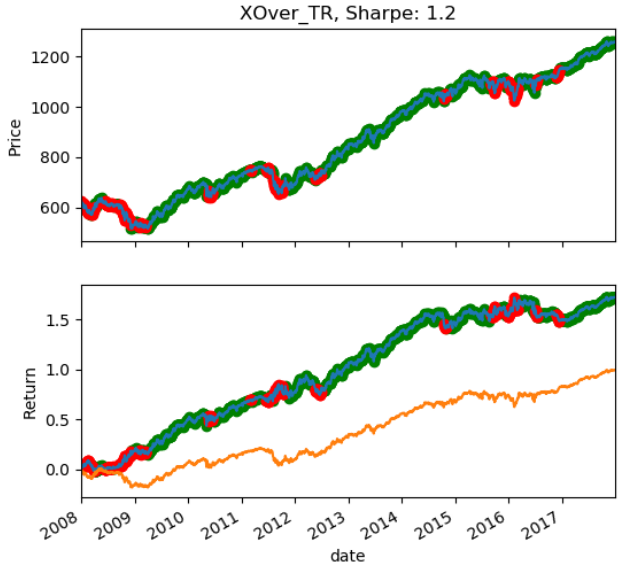
(e) XOver_TR

Figure 4.9: Results of XGBoost on the test set. Model was trained on each target time series 50 times. Plots show performance of each individual model (blue) compared to the Buy&Hold strategy (red).

Figures 4.8 and 4.9 show the individual Sharpe ratios for each test on each time series for FFNN and XGBoost models. Although models manage to beat the Buy&Hold strategy on some time series, there are some outliers that have extremely bad performance. In practise this could be catastrophic, so some sort of model averaging would be wise.



(a) FFNN



(b) XGBoost

Figure 4.10: Results of FFNN and XGBoost on XOver_TR. The plots show price and returns of the median model compared to the performance of the Buy&Hold strategy (orange). Green indicates model predicted "long" and red indicates model predicted "short".

A specific test on the XOver_TR time series can be seen in Figure 4.10 where the results are shown for both the median FFNN and XGBoost models. Both models beat the Buy&Hold strategy during year 2008, and always stay above Buy&Hold after that. The FFNN model tends to go short a lot more often than the XGBoost model, which causes it to perform worse than Buy&Hold during years where the price just keeps going up. The years 2013 and 2017 are examples of such years. It is interesting to note that both models correctly predicted the Lehman Brothers crash in 2008, and the XGBoost model also correctly predicted the 2011 crash. See Figures 4.11 and 4.12.

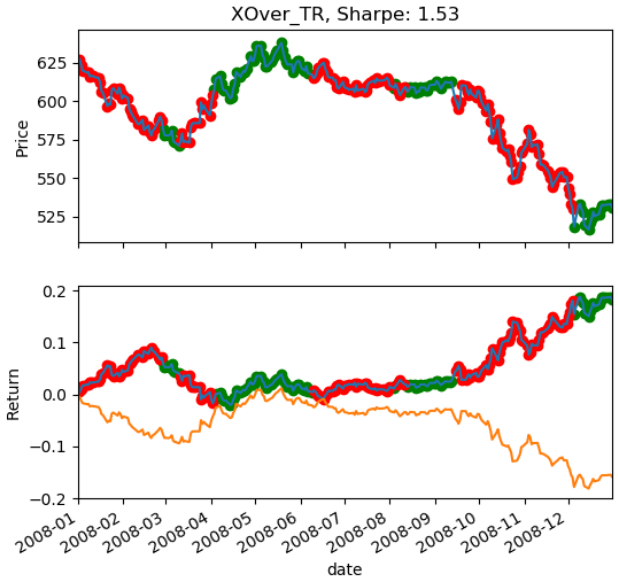


Figure 4.11: Performance of the median XGBoost model on XOver_TR, during the year 2008.

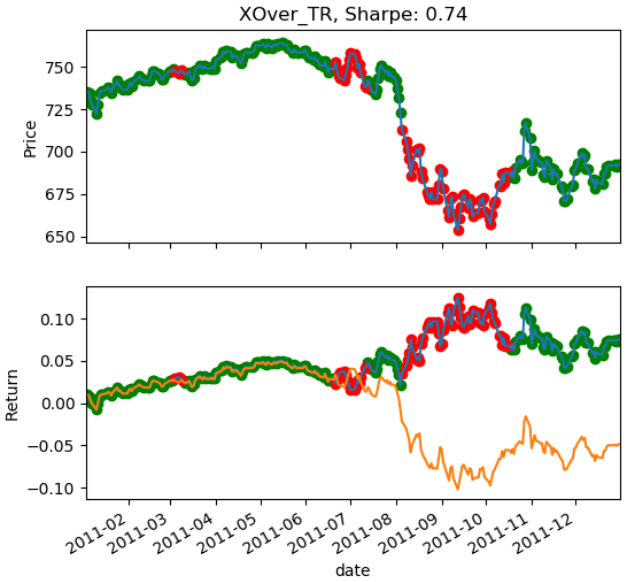


Figure 4.12: Performance of the median XGBoost model on XOver_TR, during the year 2011.

Chapter 5

Conclusions

As seen in Table 4.6, the results not only depend on the model, but also the target time series. Our experiments showed that all models had troubles with both the US and EU investment grade bond indexes. This can be due to biases in the used data (Table 3.1), or that those markets are more efficient. In practise, it would be a good idea to run the models on several markets, and choose to engage the markets with the best performance on some hold out set.

Predicting n days ahead yielded improved results over one day ahead prediction. By setting $n > 1$ we filter out the small price movements and the model acts like an ensemble of n models.

Feature selection in the form of PCA turned out to be very important for the FFNN model, but the same procedure worsened the results for the XGBoost model.

In conclustion, the XGBoost and FFNN models both managed to beat the Buy&Hold strategy on the majority of time series they were tested on. XGBoost had on average better results, both in terms of mean Sharpe ratio and standard deviation. LSTMs did not work at all, and the results for Q-learning were inconclusive. In practise, the author recommends XGBoost as a high yield bond index prediction model.

Future Work

There are many possible continuations of the work presented in this report. The Q-learning model used was not tuned properly and only gave predictions for one day ahead. It could be worthwhile to inves-

tigate how the results may change with n day ahead predictions with the Q-Network.

Another possible extension is to do more frequent train/test breakpoints. Instead of applying the process of Table 3.4 in steps of one year, one could try to apply it every half year, or even more frequently.

The parameters for the features that were used in this report, presented in Table 3.2, were selected without any validation. A more rigorous feature tuning process could improve results. The feature selection process for FFNN could also be improved. In this report we use only PCA as a form of feature selection. Perhaps it could be worth considering to train a XGBoost model, and take the most relevant features, and train a FFNN using only those features.

Some way to increase the volume of training data could also be a good idea. This can be done by either simulating data, or making use of transfer learning [20]. With transfer learning one can train a network on different datasets, and then train it on the target dataset. The idea is, if the different datasets are similar in some way, the network will learn to extract useful features that can then be applied to the target dataset.

Finally, the model validation process could be improved. Currently, for each parameter we take the value that has the best average performance on the validation set. This may be incorrect due to the fact that some parameters are dependant on one another. For example, the shrinkage parameter for XGBoost is very dependant on the maximum number of estimators. One suggestion for improvement is to run more iterations of the random search, and select the set of parameters with the best average performance.

Bibliography

[1] M. Kolanovic and R. T. Krishnamachari. Big data and ai strategies, machine learning and alternative data approach to investing. *J. P. Morgan*, 2017.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[4] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[5] Z. Zhao J. Barry J. Hunter D. Sarkar P. A. White A. Tepper L. Y. Chang M. Salem, J. Younger. Do androids dream of electric sheep? machine learning in interest rate markets. *J. P. Morgan*, 2017.

[6] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[7] S. Ramegowda S. Ungari and J. Turc. Quant market pulse version 2.0. *Societe Generale, Cross Asset Research*, 2013.

[8] Kris Longmore. Machine learning for financial prediction: experimentation with david aronson's latest work – part 1. `https://robotwealth.com/machine-learning-financial-prediction-david-aronson/`, 2016.

[9] Kris Longmore. Machine learning for financial prediction: experimentation with david aronson's latest work – part 2. https://robotwealth.com/machine-learning-for-financial-prediction-experimentation-with-da 2016.

[10] Matthew Dixon, Diego Klabjan, and Jin Hoon Bang. Classification-based financial markets prediction using deep neural networks. *Algorithmic Finance*, (Preprint):1–11, 2016.

[11] David W Lu. Agent inspired trading using recurrent reinforcement learning and lstm neural networks. *arXiv preprint arXiv:1707.07338*, 2017.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

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

[14] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[18] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey

Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning.

[19] Wes McKinney. pandas: a foundational python library for data analysis and statistics.

[20] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.