Southern University of Denmark,
Odense
Faculty of Business and Social
Sciences & Faculty of Science

June 1, 2023
Total characters: 189,348

Prediction of Stock Returns using Deep Neural Network

An Implementation of a Recurrent Neural Network

Authors:
Andreas Thrane Bruun
01-07-1997 cand.scient.oecon

Oskar Lenler Enkegaard
27-10-1997 cand.scient.oecon

Supervisors:
Aritra Dutta
Department of Mathematics
and Computer Science

Aniruddha Dutta
University of California, Berkeley

"Det erklæres herved på tro og love, at undertegnede egenhændigt og selvstændigt har udformet denne rapport. Alle citater i teksten er markeret som sådanne, og rapporten eller dele af den har ikke tidligere været fremlagt i anden bedømmelsessammenhæng."

"We hereby solemnly declare that we have personally and independently prepared this paper. All quotations in the text have been marked as such, and the paper or considerable parts of it have not previously been subject to any examination or assessment."

Andreas T. Bruun	01-06-2023
Andreas Thrane Bruun	Date
[illegible]	01-06-2023
Oskar Lenler Enkegaard	Date

Abstract

Return prediction is quite a difficult task since returns are stationary which is why they have no clear indicator as opposed to stock prices, which are non-stationary. Thus we propose a deep factor model using the recurrent neural network (RNN) architecture with both company fundamentals and macroeconomic factors. When training the neural network we use a sliding window approach to create a unique model for every month, which minimizes the validation set's mean square error. The main objective is to predict monthly stock log returns on the Nasdaq 100 index constituents and use these predictions as a basis for constructing two buy-and-hold and two long/short portfolios that aim to outperform the Nasdaq 100 index in three separate periods from 2005-2022. We find that the buy-and-hold portfolios outperform the Nasdaq 100 index on returns, volatility, and Sharpe ratio. Additionally, we find the long/short portfolios to be the most profitable but also the most volatile portfolios, showing the best performance when the market is fluctuating. Finally, we investigate the use of different hyperparameters during different economical states. Analysis of obtained results indicates that during bad economical states, more hidden layers, higher batch size, and smaller learning rate is needed to obtain a more accurate model.

Contents

1 Introduction

Predicting stock returns is a very popular area, both within research but also among investors, hedge funds, etc. To make these predictions, one must be able to process and use complex financial data driven by several factors, be it volatility, fundamental indicators, macroeconomic situations, etc. This is an area where traditional methods like technical analysis struggle. One study shows that in a total of 95 modern studies, 56 studies find positive results regarding technical trading strategies, 20 studies show negative results and 19 studies indicate mixed results (Park and Irwin, 2007). Another study finds that technical analysis does not work well when applied to the vast majority of stocks. This result is robust to different periods and different markets such as NYSE and NASDAQ (Marshall et al., 2009). Furthermore, the efficient market hypothesis suggests that the price of a stock reflects all available information about the stock(Fama, 1970). Therefore, according to this hypothesis, methods like technical analysis will not be sufficient. In addition, the random walk hypothesis suggests that "*any success in timing the market is nothing more than sheer luck*" ("Random Walk Theory", 2012, p. 106). We see that traditional methods fall short. Thus, we require more data-driven and purpose-driven forecasting techniques.

In the past years, Machine Learning (ML) has become increasingly popular in a wide range of fields. Be it natural language processing, image recognition, and of course, stock market prediction. Putting this increasing interest together with the decade-old research in stock price/return prediction,(Granger et al., 1970), it is only fitting that ML is being used in this area because of its ability to analyze large amounts of data, and if tuned correctly, give accurate predictions. ML is a wide concept with many different methods involved. In this thesis, we will be using the class of ML known as artificial neural networks (ANNs), which are models built to mimic learning by a biological brain with connected units functioning sort of like neurons in a brain, (Yegnanarayana, 2009). A class of this method is the recurrent neural network (RNN), based on the feedforward method, where the output from some nodes can contribute to the input in the next node (Salem, 2022). Therefore we want to use RNN as a prediction tool in this thesis where we examine the following research questions:

1. Is it possible to predict monthly stock log returns on the Nasdaq 100 index from 2005 until 2022?
2. Using the predicted stock log returns, can we construct a buy-and-hold portfolio that outperforms the Nasdaq 100 index from 2005 until 2022?
3. Using the predicted stock log returns, how does a long/short portfolio perform in different economical periods?
4. Which hyperparameters are efficient during different economical states?

This thesis aims to predict stock log returns in the Nasdaq 100 index (NASDAQ, 2023a) from 2005 until 2023. From the results, we construct two different portfolio strategies, on the constituents of the Nasdaq 100 index, whose performances are compared to the Nasdaq 100 index. Since our prediction period spans such an extended period, our portfolios are exposed to both bull and bear markets, allowing us to examine how the different portfolio strategies perform. We construct a deep factor model using RNN on a custom dataset containing company fundamental- and macroeconomic factors. The RNN uses a sliding window technique, which allows the model to change hyperparameters within every three months. This allows us to explore how different hyperparameters are needed during good and bad financial periods.

Organization of the thesis. The rest of this thesis is structured as follows: Section 2 details the works in the literature that are related to methods we use in our thesis, things we examine in our thesis, and other factors that might relate. Section 3 gives an in-depth walk-through of the general methodology regarding ML, deep learning, and the specific model we use in this thesis. In section 4 we explain how we collect the data used for the thesis, and how we construct our dataset - going into detail about the factors included and how they hopefully contribute to the model. Additionally, we discuss some issues regarding the dataset and go through the process of preparing the dataset for use in our model. In section 5 we set up our model, explaining how we do the model training and testing. We also show hyperparameter tuning, and exactly how our model architecture is set up with code snippets. We also go through the preparation of the different portfolios that we will be using. Section 6 goes through the results of our thesis, being both results on how the trading strategies perform and how our model performs. In section 7 we discuss different difficulties and choices within the thesis and finally, in section 8 we conclude on our proposed research questions.

The data, RNN implementation and portfolio calculations can be viewed in our GitHub repository.[1]

[1] https://github.com/Bruun97/Prediction-of-Stock-Returns-using-Deep-Neural-Network.git

2 Related work

Prediction of stock/crypto returns and prices using basic machine learning (ML) techniques is something that has been done numerous times in the past. Samuel (1959) is thought to be the first to explore the field of machine learning, and laid the groundwork for what we know as machine learning today. Hutchinson et al. (1994) propose an approach to pricing and hedging securities via learning networks. Furthermore, Kim (2003) apply support vector machines to predict the S&P 500 stock index.

However, due to the advent of digitized data, computing power, and sophisticated ML models (e.g., deep neural network (DNN) models), the last decade witnessed a surge in advanced ML techniques in stock price prediction. In this section, we will explore other works in the literature, that are related to the work we do in this thesis.

Among the very first line of works using sophisticated DNN-based ML techniques, **Stock market index prediction using artificial neural network** by Moghaddam et al. (2016) investigates the ability of DNNs to forecast the daily Nasdaq stock exchange rate. The authors train multiple feed-forward neural networks and use short-term historical stock prices as well as the day of the week as input. They use 70 days for training the model and 29 days for testing. In terms of the historical prices, the authors consider two different input sets. One with four prior days of prices and one with nine. Here, they find that different models perform better depending on the input set and that the optimal models become quite accurate (R^2 value of 0.94 or more).

A hierarchical Deep neural network design for stock returns prediction by Lachiheb and Gouider (2018) predicts 5 minutes returns on the TUNINDEX.[2] The training set contains 4 years and 3 months of data whereas the test set contains 9 months of data. They set up 4 different machine learning models, an ANN, and 3 deep models. By comparing the mean square error from each model in different business sectors, they conclude ANN performs the worst since the model needs multiple layers to identify the complex pattern of stock price correlation. With this in mind, the models are used for classifying binary stock returns where positive returns are an up prediction, and negative returns are a down prediction. Here, ANN still has the worst performance (best prediction is around 62% accuracy) whereas the deep models perform better in almost every case (best prediction is around 73% accuracy).

In **Deep Learning for Forecasting Stock Returns in the Cross-Section,** Abe and Nakayama (2018) present an implementation of a DNN model for forecasting one month ahead stock returns. For this purpose, a dataset of MSCI Japan index is prepared, containing 319 constituents in 2017. In addition, 25 factors are created and used as features in the machine learning models. The deep model uses a sliding window approach hence, after

[2] Tunisian main stock market index, including 75 stocks from all business sectors.

forecasting $[t + 1]$ the model is updated before forecasting $[t + 2]$ etc. It trains for 120 months before starting the testing period which contains 180 months of forecasting from 2002 to 2016. An equally weighted long/short portfolio is constructed which has a zero net spend that goes long in the (tertile/ quintile) undervalued stocks and short in the (tertile/ quintile) overvalued stocks. Using this setup 8 neural networks, 8 deep neural networks, and two statistical models are compared. They find that more layers in the deep models increase the accuracy of the stock returns. Furthermore, they find that 4 of the deep models outperform the statistical models. Looking at the long/short portfolios, the DNN models perform better than the other models, but only slightly.

Predicting the daily return direction of the stock market using hybrid machine learning algorithms by Zhong and Enke (2019) forecasts the daily S&P 500 ETF return direction for the next day using 60 financial and economic features. The authors forecast an up or down movement instead of a level forecast since this study's objective mainly is to develop a model with a high classification accuracy and a model that can be used in a practical trading environment, where direction forecasts often lead to more profitable investment strategies. In this study, the authors use both deep neural networks (DNNs) and artificial neural networks (ANNs) both designed to be effective at pattern recognition (DNN with many hidden layers and ANN with 10 hidden layers). The trading strategy entails going long in the S&P 500 if an up movement is predicted, or going long in one-month T-bills. Return, volatility, and Sharpe ratio are calculated, and the authors find that trading strategies based on DNN predictions are generally more profitable in terms of Sharpe ratio.

In **Deep Learning with long short-term memory networks for financial market predictions** by Fischer and Krauss (2018) presents an LSTM model which predicts directional movements on the S&P500 index. To indicate the strength of LSTM, they compare the model to other models, a Random Forrest, a Deep neural network, and a logistic regression classifier. The LSTM outperforms these other model types from 1990 to 2009, and has a significant daily return of 0.46% prior to transaction cost. From 2010 to 2015 the model generates a daily return of around zero. Furthermore a long/short portfolio is introduced which buys short-term losers and sells short-term winners. This yields a daily return of 0.23% prior to the transaction cost.

The works that are closely related to the work presented in this thesis are **Deep Factor Model** by Nakagawa et al. (2018) and **Deep Fundamental Factor Models** by Dixon and Polson (2020).

Deep Factor Model presents a return model and a risk model in an unified framework. That is, they use deep neural network (DNN) model to predict stock returns using various factors, and then use a method called "layer-wise relevance propagation" to determine which factors contribute to predictions. This approach is useful in multiple ways. Firstly, if one is able to find out the probable future value of a factor, it is possible to construct a portfolio that favors the good factors and avoids the bad. Secondly, by capturing the correlation among the factors, one can construct a balanced portfolio that diversifies risk away. When training the models, the authors use a sliding window, where they train on the last 60 sets of training data, and then slide one month ahead, to carry out a monthly forecast. This gives the model plenty of data in each window to carry out forecasts. The models created are tested on the Japanese stock market on TOPIX[3] constituents, where the authors find that deep models outperform a baseline linear model, a support vector regression model, and a random forest model on profitability both in terms of return and Sharpe ratio and on accuracy in terms of mean absolute error (MAE) and root mean squared error (RMSE). Additionally, they find that a shallow deep model (with fewer layers) has the best prediction accuracy in terms of MAE and RMSE while the deeper model (with more layers) is the most profitable in terms of Sharpe ratio. The fact that deep models outperform the other models implies, that the relationship between stock returns and factors is nonlinear, and a model that can capture this non-linearity is superior.

Deep Fundamental Factor Models present a framework for models, developed to capture non-linearity and interaction effects in factor modeling. An essential aspect of the paper is the interpretability. As stated in the paper, neural networks are like "black boxes" and thus, their behavior can not be reasoned on statistical grounds. This is what the authors aim to do. They develop a deep fundamental factor model with 50 factors for 3290 stocks all indexed on the Russell 1000 index over a 30-year period. This model is then compared to OLS- and LASSO-based factor models based on performance and factor interpretability. In the first month, the model is fitted to the factor exposures and monthly excess returns of the next period. Then cross-validation is performed to tune the model, and the model can then be applied to the factor exposures in [t+1] to predict excess monthly returns over [t+1,t+2]. Finally, to evaluate the model in comparison to linear and quadratic regressions, an equal-weighted portfolio of n stocks with the highest predicted returns is created. The information ratios are then compared, with the Russell 1000 index as a benchmark. Here, DNN models outperform the other models. For interpretability, the authors are able to find out which factors are preferred by the models, and how the neural network is more sensitive to the factors, meaning it is better at capturing outliers.

[3] Tokyo Stock Price Index, composed of top domestic companies on the Tokyo Stock Exchange ("TOPIX", 2023).

3 Methodology

In this section, we will go through the methodology and some important aspects, that we use in this thesis (Goodfellow et al., 2016). Firstly, we go through the very basics of machine learning. In this we highlight some core concepts that will help us understand how we can use this methodology later. Additionally, we consider two important optimization algorithms. Next, we move to the concept of *deep learning*, where we go into detail about how this concept works, and highlight some possible issues that we might run into with our own work using these methods. Lastly, we go through the specific model we will base our work on, and an important extension that we will not explicitly use in this thesis, but would like to base some future work upon.

3.1 Basics of Machine learning

To truly understand the concept of Deep learning, which is the machine learning technique we will use to predict stock log returns, one first needs to understand the concepts of Machine learning. According to Goodfellow et al. (2016), Machine learning aims to find patterns, fit data, etc. Machine learning uses an algorithm which is called a learning algorithm. This algorithm has the ability to learn from input data. Using this algorithm, we have the possibility to solve complex tasks which humans cannot solve. To solve this, one could present the data as $\boldsymbol{x} \in \mathcal{R}^n$, where $\boldsymbol{x}$ represents a vector of features. These features are not what needs to be predicted, but data from which the algorithm learns behavior.

Here are two examples of tasks a machine-learning algorithm could solve:

- **Classification**: The algorithm categorizes the data into different categories. The algorithm maps the input to a category $f : \mathbb{R}^n \to \{1, ..., k\}, \quad y = f(\boldsymbol{x})$. Using this approach, one could afterward calculate the accuracy of the classification. This accuracy parameter is calculated by comparing correct results with the algorithm's results.

- **Regression**: The algorithm returns a numerical value $f : \mathbb{R}^n \to \mathbb{R}$. Since it provides a numerical value, it makes no sense to calculate the accuracy. Therefore it is often more optimal to calculate the mean square error (MSE) to determine the precision. See equation 1.

Furthermore, when examining learning algorithms, two general ways exist to explore the dataset.

An *unsupervised learning algorithm* learns useful properties of the dataset structure. This can be used when performing clustering. Here the data is divided into "groups" with similar results. This could be stock returns in the same region etc. An easy way to understand the unsupervised concept is to think of unsupervised learning. Here the student must provide meaning to the given material without guidance.

On the other hand, a *supervised learning algorithm* experiences datasets where each feature is associated with a target. The features could be total sales, total earnings, etc. Here the target could be the stock log return. This can be described as a conditional probability $p(\boldsymbol{y}|\boldsymbol{x})$. When preparing a dataset, one must design a matrix $\boldsymbol{X} \in \mathbb{R}^{\text{datapoints}\times\text{features}}$.

3.1.1 Linear Regression

Goodfellow et al. (2016) define a very simple learning algorithm namely the linear regression to understand the properties of a machine learning algorithm. This model predicts $y \in \mathbb{R}$ using some input vector $\boldsymbol{x} \in \mathbb{R}^n$. Hence,

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}.$$

One could think of $\boldsymbol{w}$ being weights. The weights determine how important each feature is. A feature with a positive weight will increase our prediction value and vice versa. We first train the model on a training set and test the model on a test set. After setting up the dataset design matrix, one could evaluate accuracy by regression using MSE,

$$\text{MSE} = \frac{\sum_i (\hat{\boldsymbol{y}}^{(\text{test})} - \boldsymbol{y}^{(\text{test})})_i^2}{m}, \tag{1}$$

where $\hat{\boldsymbol{y}}$ is the algorithms' predictions, and $\boldsymbol{y}$ is the true value. Now, the machine learning algorithm needs to minimize the MSE. This could be done by minimizing the MSE in the training set and applying the weights to the test set. Furthermore, we can introduce a bias parameter b of the affine transformation,

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + b.$$

This allows mapping from features to prediction to be an affine function.

3.1.2 Underfitting, Overfitting, and Capacity

The ultimate goal of machine learning is to minimize the generalization error which is the test set error, without the model ever viewing the test set (Goodfellow et al., 2016). This is also what differentiates machine learning from optimization since the model cannot use any optimization on the test set. The big question is: Is it even possible to minimize the error on the test set when only the training set is observed? This is possible if we assume that the dataset is i.i.d. Hence, the test and train sets are identically distributed. We can denote the probability distribution as p_{data}, which means the training- and test examples use the same distribution. Naturally,

$$\mathbb{E}(\text{Error}_{\text{train}}) \leq \mathbb{E}(\text{Error}_{\text{test}}),$$

since both expectations are computed using the same dataset sample process. If we want a small expected error in the test set, we would need to lower the expected error in the training set. Afterward, the goal is to minimize the gap between the training and test errors. This comes with some challenges which can be described as **under- and overfitting**.

- **Underfitting** arises when the training error is too big which results in the test error being big.
- **Overfitting** arises when the training error becomes very small and the test error begins to increase.

To control whether under/overfitting happens, we introduce **capacity**. The capacity can be controlled using a hypothesis space, which is the set of functions the model is allowed to use when fitting the data. This could be allowing the model to fit a linear function, quadratic function, and polynomial with a high degree. It is worth noticing that when the capacity is too complex, the model will overfit, as well as when the capacity is too simple, the model will underfit. Overfitting leads to a gap between training error and test error as previously mentioned. Thus, the optimal capacity arises when the true complexity is provided.

There are multiple ways of changing the model's capacity (Goodfellow et al., 2016). The representational capacity is the family of functions the learning algorithm chooses from when reducing the training error. The algorithm finds a function, within the family, which reduced the training error more than the others. This might not be the optimal function, since it can be close to impossible to find. The chosen capacity is called the effective capacity. Furthermore, as earlier described, to obtain a very low generalization error, we need to find a sufficient complex hypothesis space. When looking at figure 3.1, there is a clear link between capacity and error size. When the capacity is too low, the model is underfitting hence, it has a large training and generalization error. When capacity is too high, we observe the training error keeps decreasing while the generalization error increases. This leads to overfitting since the gap between the errors is increasing. Besides changing the learning algorithm's hypothesis space, there are other ways to modify it. One could think of an example where the model tries to predict $f(x) = e^x$. Here we could control the function which the learning algorithm uses. This is to prefer $f(x)$ over other functions. If one function is preferred over another this will always be selected if it minimizes the error equally. But in case it does not, the other function will still be chosen. Hence, looking at a linear function we can introduce a weight decay

$$j(\boldsymbol{w}) = \text{MSE}_{\text{train}} + \lambda \boldsymbol{w}^\top \boldsymbol{w}.$$

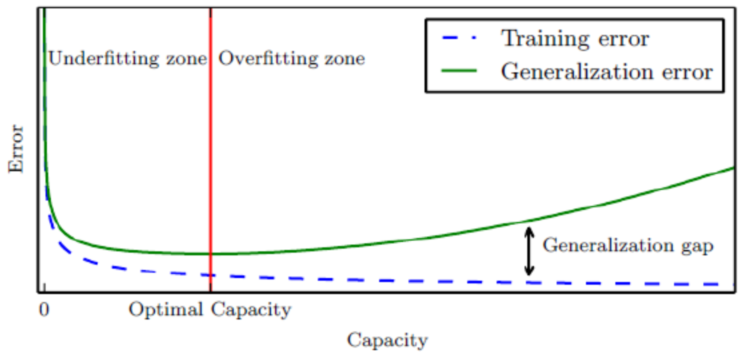


Figure 3.1: *Relationship between capacity and error. Source: Figure 5.3 (Goodfellow et al., 2016).*

here λ controls our preference for smaller weights in the next time periods. It is clear that this chooses the optimal best payoff between fitting and being small hence, some of the features are ruled out since they would have weights equal to zero.

These modifications are called regularization which will be further explained in Section 3.2.2. The idea is to modify the learning algorithm to decrease the generalization error and to keep the training error unchanged (Goodfellow et al., 2016).

3.1.3 Hyperparameters and Validation Sets

Hyperparameters are parameters that are not estimated/ determined by the learning algorithm. The hyperparameters choose the behavior of the learning algorithm. In the section “Capacity, Over- and Underfitting” we encountered some types of hyperparameters e.g. degree of a polynomial and λ.

According to Goodfellow et al. (2016) hyperparameters are often difficult to estimate, especially those which affect the capacity. Thus, the hyperparameters are prefixed since the learning algorithm would increase to maximum model capacity, and thereby overfitting, if the hyperparameters were chosen inside the model. This is the main reason why a validation set is needed. A validation set is a subset of the training set that the training algorithm does not observe. More generally, the training data is split into two disjoint sets. One of the subsets is our standard training set which predicts parameters. The other subset is the validations set. From the validation set, we estimate the generalization error. The general idea can be shown in Figure 3.2. First, we train the model using the train set. After every epoch, we evaluate the model using the validation set. This will tweak the model such that we get better results. This goes on until the training phase is over. We can now choose the best model, the model with the lowest validation set error, to run on the test set. Since the model never trains on the validation set, this is a very good way to secure that the model is not overfitting too much. But since the model is tweaked accordingly to the validation set, the validation set typically underestimates the generalization error.

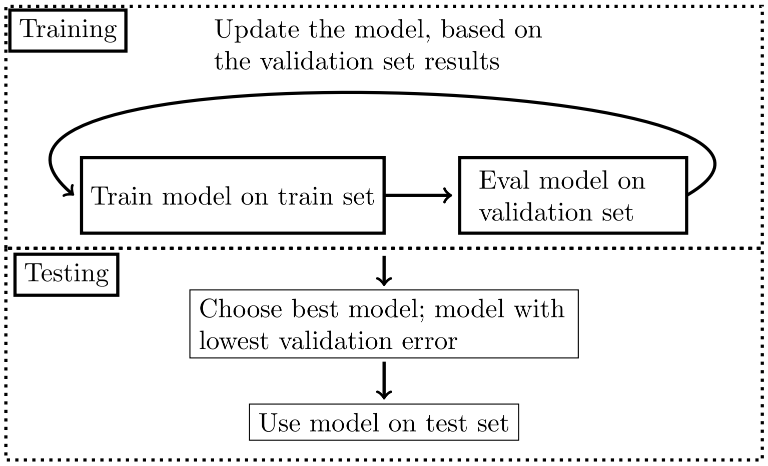

Training
Update the model, based on the validation set results
Train model on train set
Eval model on validation set
Testing
Choose best model; model with lowest validation error
Use model on test set

Figure 3.2: *Illustration of validation set. Source: Own work.*

3.1.4 Tools from statistics

When looking upon the machine-learning goal of solving certain tasks, we can look to statistics for some tools that can help us achieve our goals. Earlier we talked about generalization, underfitting, and overfitting, and to characterize these concepts, we can use estimators, bias, and variance described by Goodfellow et al. (2016).

Let us first consider **point estimation**. This is the concept of providing the best prediction of what we are interested in examining. This can be one value or a vector of values such as the weights in our linear regression example in section 3.1.1. When looking at estimations versus true values, we need a way to discern what we are talking about. We can do this by denoting the parameter $\boldsymbol{\theta}$ and the point estimate $\hat{\boldsymbol{\theta}}$. Now, we can let a set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$, be a set of m independent and identically distributed data points. Then, a point estimator is:

$$\hat{\boldsymbol{\theta}}_m = g(\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}).$$

Now, the above definition does not require that the function g returns a value close to what we are trying to estimate, that is, the true $\boldsymbol{\theta}$. This gives great flexibility since almost any function thus can work as an estimator. It is worth noting though, that a good estimator is one whose output is close to the true $\boldsymbol{\theta}$.

Another case of point estimation is when one tries to estimate the relationship between some input and target variables. As is the case for this thesis. This is called **function estimators**.

The concept of function estimation is widely the same as point estimation. Here we are interested in predicting a variable $\boldsymbol{y}$ given an input vector $\boldsymbol{x}$. As is the case for the function g in point estimation, we here assume that there is a function $f(\boldsymbol{x})$ that describes the relationship between $\boldsymbol{y}$ and $\boldsymbol{x}$. An example of this could be $\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon}$ is some part of $\boldsymbol{y}$ that cannot be estimated by f from $\boldsymbol{x}$.

Next, we are interested in looking at **bias**. The bias of an estimator is defined as

$$\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta}, \tag{2}$$

where the expectation is over the data. If $\text{bias}(\hat{\boldsymbol{\theta}}_m) = 0$, the estimator, $\hat{\boldsymbol{\theta}}_m$ is said to be unbiased, since this implies that $\mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$. Additionally, the estimator is said to be asymptotically unbiased if $\lim_{m\to\infty} \text{bias}(\hat{\boldsymbol{\theta}}_m) = 0$, since this implies that $\lim_{m\to\infty} \mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$.

Example: Consider a set of samples $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ from a Gaussian distribution with $p(x^{(i)}) = \mathcal{N}(x^{(i)}; \mu, \sigma^2)$, where $i \in \{0, ..., m\}$. Now, first, let us estimate the mean. A common estimator of the Gaussian mean is the sample mean $\hat{\mu}_m = 1/m \sum_{i=1}^{m} x^{(i)}$. Inserting this in (2), we get

$$\begin{aligned}
\text{bias}(\hat{\mu}_m) &= \mathbb{E}(\hat{\mu}_m) - \mu \\
&= \underbrace{\mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right]}_{=\mu} - \mu \\
&= 0.
\end{aligned}$$

Thus we find, that the sample mean is an unbiased estimator of the mean. Next, we can try to estimate the variance of the Gaussian distribution. Here, we consider the sample variance as an estimator $\hat{\sigma}_m = 1/m \sum_{i=1}^{m} \left(x^{(i)} - \hat{\mu}_m\right)^2$. Again, we insert in (2)

$$\begin{aligned}
\text{bias}(\hat{\sigma}_m) &= \mathbb{E}(\hat{\sigma}_m) - \sigma^2 \\
&= \underbrace{\mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} \left(x^{(i)} - \hat{\mu}_m\right)^2\right]}_{=\frac{m-1}{m}\sigma^2} - \sigma^2 \\
&= -\frac{\sigma^2}{m}.
\end{aligned}$$

Here, we find that the sample variance is actually a biased estimator of the variance.

Speaking of variance, another thing regarding the estimators, that we might want to consider is how much they vary. For this, we can examine the **variance**, $\mathrm{Var}(\hat{\theta})$. Sometimes we consider the square root of this variance as well, called the standard error, or $\mathrm{SE}(\hat{\theta})$. These measures provide insight into how we expect the estimator to vary given how we assemble the dataset. Just as we would want an estimator to have a low bias, we would also like the estimator to have a low variance. The square error of the mean is:

$$\mathrm{SE}(\hat{\mu}_m) = \sqrt{\mathrm{Var}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}}. \tag{3}$$

The SE of the mean is very useful in machine learning. Often, the generalization error is estimated by computing the sample mean of the error on the test set. The accuracy of this varies however, since it is dependent on the number of samples in the test set.

We have now talked about two types of errors in estimators. Bias and variance. Bias being the expected deviation from the true value of what we are trying to estimate and variance being the amount we expect to vary from the true value given any sampling of the data. But how can one choose between these types of errors? Say that we are given the choice between a model with a large bias or one with a large variance. A good choice for this is using mean squared error (MSE) and comparing this between the estimates. MSE is given by:

$$\mathrm{MSE} = \mathbb{E}[(\hat{\theta}_m - \theta)^2] = \mathrm{Bias}(\hat{\theta}_m)^2 + \mathrm{Var}(\hat{\theta}_m).$$

As one can see, this MSE incorporates both bias and variance and measures the estimator's overall expected deviation from the actual value of $\boldsymbol{\theta}$. That is, a good estimator is one whose MSE is relatively low. These measures, bias and variance, and their relationship are well related to some concepts discussed in earlier sections. Here we are talking about capacity, underfitting, and overfitting. If generalization error is measured by MSE, a larger capacity tends to result in a larger variance and smaller bias. This is illustrated in Figure 3.3, where we clearly see the relationship between the capacity of the model, and both the bias and variance. We notice that just like in figure 3.1, the generalization error follows a U-shape, and has a sweet spot in capacity, where there is no under- or overfitting.

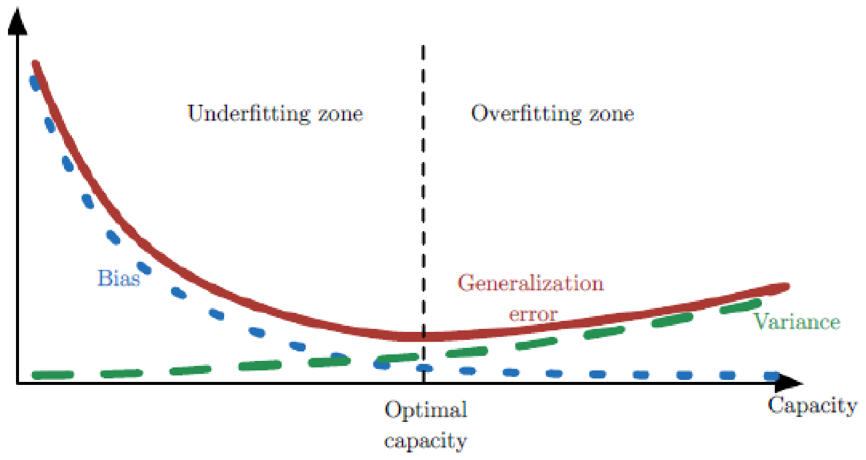


Figure 3.3: *Relationship between capacity and bias/variance. Source: Figure 5.6 (Goodfellow et al., 2016).*

3.1.5 The Kernel Trick

Earlier we mentioned supervised learning algorithms, which brings us to one of the most influential approaches to this type of learning; support vector machines (SVM). We will not spend much time on this topic but will note, that SVM is very similar to that of regression, described in section 3.1.1, since it is driven by the similar linear function $\boldsymbol{w}^\top \boldsymbol{x} + b$. What we will spend some time on, however, is the **kernel trick** (Goodfellow et al., 2016).

This trick has its basis in the observation that many machine learning approaches can be written in terms of dot products. As an example, it can be shown that the linear function used in SVM can be written as

$$\boldsymbol{w}^\top \boldsymbol{x} + b = b + \sum_{i=1}^{m} \alpha_i \boldsymbol{x}^\top \boldsymbol{x}^{(i)},$$

where $\boldsymbol{x}^{(i)}$ is a training example and α is a vector of coefficients. Thus, we can replace the $\boldsymbol{x}$ by the output of some given feature function $\phi(\boldsymbol{x})$ and the dot product with some function $k(\boldsymbol{x}, \boldsymbol{x}^{(i)}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{x}^{(i)})$ which is called a **kernel**. So after replacing the dot product with these kernel observations, we make predictions using the function

$$f(\boldsymbol{x}) = b + \sum_{i} \alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)}).$$

The function described above is nonlinear with respect to $\boldsymbol{x}$, however, the relationship between $\phi(\boldsymbol{x})$ and $f(\boldsymbol{x})$ as well as the relationship between α and $f(\boldsymbol{x})$ is linear. An interesting observation is, that this kernel-based function is equivalent to applying $\phi(\boldsymbol{x})$ to all inputs, and then learning a linear model in in this new transformed space.

The kernel trick is useful for multiple reasons. Firstly, it allows us to learn nonlinear models (as functions of $\boldsymbol{x}$) using techniques that are guaranteed to converge efficiently. Second, the kernel function, k, makes the implementation significantly more computationally efficient than constructing two $\phi(\boldsymbol{x})$ vectors and taking their dot product. It is possible, that $\phi(\boldsymbol{x})$ in some cases is infinite dimensional. This would of course result in an infinite computational cost for explicitly calculating the dot product. Often, $k(\boldsymbol{x}, \boldsymbol{x}')$ is a nonlinear, but tractable, function of $\boldsymbol{x}$, even if $\phi(\boldsymbol{x})$ is intractable. As an example of an infinite-dimensional feature space with a tractable kernel, one can consider a feature mapping $\phi(x)$ over the non-negative intergers x. If this mapping returns a vector containing x ones followed by infinitely many zeros, we can write a kernel function $k(x, x^{(i)}) = \min(x, x^{(i)})$. This function is equivalent to the corresponding infinite-dimensional dot product.

SVM is not the only method enhanced by using the kernel trick. This can be done to many methods, that we can then classify as kernel machines. A large drawback to kernel machines is that they are subject to very high computationl costs of training, when the dataset is large. Another problem is that kernel machines with simple kernels have a hard time generalizing. This is where deep learning, which we will discuss later, comes into the picture, as these methods are designed to overcome these limitations of kernel machines.

3.1.6 Stochastic Gradient Descent

A very important optimization method in machine learning, and more importantly, deep learning, is stochastic gradient descent (SGD) (Robbins and Monro, 1951). This method is an extension of the gradient descent technique, where the goal is to optimize a function $y = f(x)$ by changing x. The original method - gradient descent, simply works by performing simple calculus. The derivative of the function is of course denoted as $f'(x)$ or as $\frac{dy}{dx}$. This derivative gives the slope of f at the point x, and is therefore useful for minimizing (or maximizing) a function since it tells us how to alter the variable x to get a change in y. More generally, we know that $f(x - \epsilon \cdot \text{sign}(f'(x)))$ is less than $f(x)$ for a small enough ϵ, and thus we can minimize (or maximize) $f(x)$ by moving x in small steps in the opposite direction than the sign of the derivative. This gradient descent technique is pictured in figure 3.4 below.

However, a problem arises when trying to apply this method in machine learning (Goodfellow et al., 2016). It is well known, that in machine learning, a large training set is necessary for good generalization but can quickly become very computationally expensive. The cost function used by a machine learning algorithm is often viewed as a sum over training examples of some loss function. Take for example the negative conditional log-likelihood

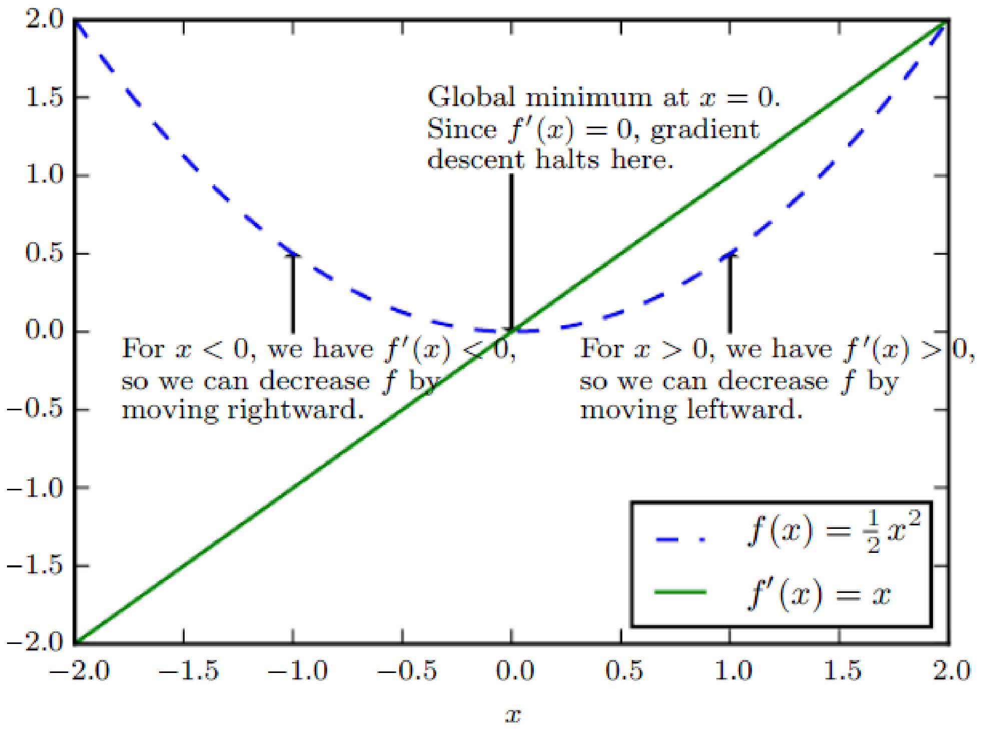


Figure 3.4: *An illustration of the gradient descent technique. Source: Figure 4.1 (Goodfellow et al. (2016)).*

of the training data. This is written as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x},y\sim\hat{p}_{\text{data}}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}).$$

Here, L is the loss $L(\boldsymbol{x}, y, \boldsymbol{\theta}) = -\log p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$. For this, gradient descent requires us to compute

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}).$$

This requires a lot of computations when the training set becomes larger, and thus gradient descent quickly becomes infeasible.

The idea behind **stochastic** gradient descent, is that the gradient is actually an expectation. Thus, this expectation can be approximated by using a small set of samples. That is, we sample a minibatch of samples $\mathbb{B} = \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m')}\}$ drawn uniformly from the training set. Thus, the minibatch sample size, m', can be a small number of examples (1 to

100's), but most importantly is kept a fixed amount as m increases. Thus, the training set could theoretically contain billions of samples, but we fit it using just a couple of hundred examples. This estimate of the gradient is then found by

$$\boldsymbol{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}),$$

and the stochastic gradient descent algorithm can follow the estimated gradient downhill just like the gradient descent algorithm would:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g},$$

where ϵ is the learning rate. In the machine learning field, it is now well known, that many models work very well when trained and optimized with (stochastic) gradient descent. While the model may not arrive at a local minimum quickly, it oftentimes finds a low value of the cost function very quickly.

Before deep learning, the primary way to learn nonlinear models was to use the kernel trick, described in 3.1.5. However, this often meant that an $m \times m$ matrix $G_{i,j} = k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)})$ was to be constructed, which is computationally very expensive. For datasets with millions, or even billions, of examples, this was far from preferable. This is why deep learning was interesting since it was able to generalize to larger datasets faster.

3.1.7 Extension: ADAM

The Adam optimizer is an extended version of SGD introduced by Kingma and Ba (2014). This section is largely based on that same paper.

Adam is a method for efficient stochastic optimization that only requires first-order gradients and has a small memory requirement. The name of the method; Adam, comes from *adaptive moment estimation* since Adam actually computes individual adaptive learning rates for its different parameters from estimates of the first and second moments of the gradients.

Adam is actually a combination of two other popular methods: AdaGrad (Duchi et al., 2011) which works well on sparse gradients and RMSProp (Tieleman and Hinton, 2012) which works well in non-stationary settings. Adam works by implementing the exponential moving average of the gradients so that it can scale the learning rate, and thus it keeps an exponentially decaying average of the past gradients. There are a few parameters that need to be set, such as the learning rate α, the exponential decay rates for moment estimates, β_1 and β_2, and a constant ϵ. Additionally, a stochastic objective function, $f(\theta)$ is needed, with parameter vector θ. The algorithm's goal is then to find the converged values of this parameter vector (weights). This is done by first initializing θ_0, and then computing the exponential moving average of the gradient (m_t) and the squared gradient (v_t), which are the estimates of the first and second moments respectively. m_t and v_t are calculated as follows:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t,$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2.$$

Above, g_t is the gradient of the stochastic objective function, $g_t = \nabla_\theta f_t(\theta_{t-1})$. Both m_t and v_t are initialized with 0-vectors ($m_0 = 0$ & $v_0 = 0$), and thus these estimates are biased around 0 since β_1 and β_2 are close to 1. To counteract this, we have to calculate the bias-corrected versions of m_t and v_t:

$$\hat{m}_t = \frac{m_t}{1-\beta_1},$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2}.$$

Finally, we can update the parameters:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}.$$

All steps above are then repeated until θ converges.

Example: Let us consider the following numerical example, to show the intuition behind the Adam optimizer.

We have the following sample dataset, where the objective is to predict the stock price given the number of sales:

Sales	60	76	85	50	55	100	96	45	78
Stock Price	172	190	170	150	161	195	187	160	178

The hypothesis function here is: $f_\theta(x) = \theta_0 + \theta_1 x$ and the cost function is $J(\theta) = \frac{1}{2}\sum_i^n (f_\theta(x_i) - y_i)^2$. Now, we calculate the gradient:

$$g_t = \nabla_\theta f_\theta(x) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \end{pmatrix}$$

We expand the cost function

$$\frac{1}{2}(\theta_0 + \theta_1 x - y)^2 = \underbrace{\frac{1}{2}\left(y^2 - 2\theta_1 xy - 2\theta_0 y + \theta_1^2 x^2 + 2\theta_0\theta_1 x + \theta_0^2\right)}_{:=(\star)},$$

and can then evaluate the derivatives:

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta_0} &= \frac{\partial}{\partial \theta_0}(\star) = -y + \theta_1 x + \theta_0 = f_\theta(x) - y,\\ \frac{\partial J(\theta)}{\partial \theta_1} &= \frac{\partial}{\partial \theta_1}(\star) = -yx + \theta_1 x^2 + \theta_0 x = (f_\theta(x) - y)x.\end{aligned}$$

Now we are ready to perform the algorithm. We set initial values $\theta = \begin{pmatrix}50\\2\end{pmatrix}$, $\alpha = 0.01$, and $\beta_1, \beta_2, \epsilon$ are set to $0.9, 0.999, 10^{-7}$ respectively. Remember that m_0 and v_0 are initialized with 0. We can then calculate m_1 and v_1:

$$\begin{aligned}m_1 &= 0.9 \cdot 0 + (1 - 0.9) \cdot \begin{pmatrix}50 + 2 \cdot 60 - 172\\(50 + 2 \cdot 60 - 172)60\end{pmatrix} = \begin{pmatrix}-0.2\\-12\end{pmatrix},\\ v_1 &= 0.999 \cdot 0 + (1 - 0.999) \cdot \begin{pmatrix}(50 + 2 \cdot 60 - 172)^2\\((50 + 2 \cdot 60 - 172)60)^2\end{pmatrix} = \begin{pmatrix}0.004\\14.4\end{pmatrix}.\end{aligned}$$

The bias-corrected versions are then

$$\begin{aligned}\hat{m}_1 &= \begin{pmatrix}-0.2\\-12\end{pmatrix}\frac{1}{1 - 0.9} = \begin{pmatrix}-2\\-120\end{pmatrix},\\ \hat{v}_1 &= \begin{pmatrix}0.004\\14.4\end{pmatrix}\frac{1}{1 - 0.999} = \begin{pmatrix}4\\14400\end{pmatrix}.\end{aligned}$$

Finally, we can update the weight parameters:

$$\theta = \begin{pmatrix}50\\2\end{pmatrix} - 0.01 \cdot \begin{pmatrix}-2\\-120\end{pmatrix}\frac{1}{\sqrt{\begin{pmatrix}4\\14400\end{pmatrix} + 10^{-7}}} = \begin{pmatrix}50.01\\2.01\end{pmatrix}.$$

This procedure is then repeated until θ is converged.

To conclude, both Adam and SGD are optimization algorithms used on a neural network during training. The objective for both algorithms of course is to update the weights of the neural network, but they have different approaches to doing so.

In general, both algorithms are suitable for predictions, and thus the choice comes down to the specific requirements of the problem at hand. Predominantly, Adam is more popular in deep learning tasks, since it converges faster and more reliably. SGD on the other hand, can be useful in cases where the dataset is small and the model is simple. The choice of which optimizer to use is then best made, after empirical evaluation of both optimizers with different sets of hyperparameters, to see which optimizer suits the model at hand best.

3.2 Deep learning

By Goodfellow et al. (2016), deep learning is a sub-branch of machine learning. As well as machine learning, deep learning uses artificial intelligence to solve complex problems. The big difference is that deep neural networks have multiple layers (depth) hence deep learning. This allows the neural network to recognize very complex patterns when provided with a vast amount of data since the layers pass on "information" within the network. This allows deep learning to learn more complex patterns than traditional machine learning methods due to its ability to use non-linear transformations on multiple layers. It has revolutionized areas within artificial intelligence such as speech recognition, visual object recognition, object detection, etc. (LeCun et al., 2015). It has also shown good promise in the field of finance (stock return prediction), where LSTM significantly outperforms other models such as random forest, a deep neural network, and a logistic regression classifier (Fischer and Krauss, 2018). In this chapter, there will be a walkthrough of some different deep-learning techniques, which are essential within the subject.

3.2.1 Deep Feedforward Networks, Hidden Units, Architecture Design, and Back-Propagation

A deep feedforward network is a network that tries to approximate a function f^*. Goodfellow et al. (2016) define it as $\boldsymbol{y} = f(\boldsymbol{x};\boldsymbol{\theta})$ where $\boldsymbol{y}$ is the output, $\boldsymbol{x}$ is the input and $\boldsymbol{\theta}$ is the estimated parameters. These networks are called feedforward networks since they evaluate $\boldsymbol{x}$ using the function f^* and output $\boldsymbol{y}$. One also has the possibility to have a feedback connection, which is used in a recurrent neural network (RNN), and provides the network with backward-moving information hence, the output is transferred back into the neural network.

One could describe the feedforward network as

$$f(\boldsymbol{x}) = f^{(3)}\left(f^{(2)}\left(f^{(1)}\left(\boldsymbol{x}\right)\right)\right),$$

where we observe that they are connected in a chain which is one of the reasons why it is called a network. We clearly see a feed-forward network since $f^{(1)}$ is to be evaluated first, then $f^{(2)}$ and finally the output layer $f^{(3)}$. The shown network has three layers and hence has a depth of three.

When using a feedforward network, our main goal during training is to make $f(\boldsymbol{x}) \to f^*(\boldsymbol{x})$. This is done by providing the network with a training set. Within the training set, there exist some features $\boldsymbol{x}$ which are used to predict the output label $\boldsymbol{y}$. By providing the model with enough data, we obtain $f^*(\boldsymbol{x}) \approx y$, and thus, the output layer needs to approximate a value for $\boldsymbol{y}$ which is close to the real values. This is only true for the output layer. For the other layers, the learning algorithm must find the optimal way to approximate f^*. These layers are often referred to as hidden layers since these do not provide any output. Furthermore, a feedforward network is a neural network hence, the network consists of a width which is the dimension within the hidden layers. These are often called neurons. These neurons/units receive inputs from many other units and use the gathered information to calculate the units' unique activation function. Using this, the network can achieve statistical generalization on the data set.

We will in the following discuss some neural network designs, which can improve the performance of the model (Goodfellow et al., 2016).

A hidden unit, also referred to as a neuron, is a unit that receives an input from the previous layer and applies a non-linear transformation which generates some outputs that are passed on to the next layer in the model. It is referred to as "hidden", since the output is not directly observable. Each hidden unit computes a single output value using a weighted sum of the inputs and finally, an activation function is applied. During the training process, the weights and biases associated with each unit, are learned using a machine-learning technique called backpropagation, which will be explained later. The question is then, why are these hidden units so important? The hidden units are hyperparameters, which means they are defined and fixed before the model is running. The hidden units can improve performance since a lot of hidden units increase the complexity of the model, which allows the model to learn complex patterns. Too many hidden units will lead to overfitting, where the model will perform very well on the training data, but not on the test data.

Even though the hidden units are of extreme importance, it can be difficult to determine the design of the hidden units. There are multiple ways to design such units. Some of the most used hidden units are the logistic sigmoid

$$g(z) = \sigma(z),$$

defined by the formula

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{e^z+1},$$

and the hyperbolic tangent activation function

$$g(z) = \tanh(z),$$

defined by the formula

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{e^{2z}-1}{e^{2z}+1}.$$

The sigmoidal units are not recommended to use in gradient-based learning since they are very sensitive to an input when z is close to zero. When the inputs are not close to zero, we would end up with the unit value being either very low or very high. In an RNN, this activation function is commonly used together with the hyperbolic tangent.

Another approach is to use the rectified linear unit (ReLU), firstly introduced by Fukushima (1975). This approach has become quite popular since they are easy to optimize. By looking at the activation function of ReLU,

$$g(z) = \max\{0, z\},$$

we observe the function returns z if $z > 0$, otherwise it returns zero. When using other activation functions like the sigmoid function, we could end up with a vanishing gradient problem. which can make it very difficult for the model to learn. For the ReLU, this is not the case since the gradients remain large as long as the units are active. When examining the affine transformation

$$\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}),$$

we could end up in a situation where some units never "activate" and thus, produce zero outputs. This problem is often referred to as dying ReLU and can be resolved by assigning small positive values to the vector $\boldsymbol{b}$.

Another very important aspect when creating a neural network is **architectural design**. This defines the structure of the network. We assign the number of units to each layer and how these units should be connected. This can be visualized in the following way

$$\begin{aligned}
\boldsymbol{h}^{(1)} &= g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right), \\
\boldsymbol{h}^{(2)} &= g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right).
\end{aligned}$$

We observe that this is a chain-based architecture since the output of the first layer is in the second layer. There is a tradeoff in choosing the depth of the model. Having a few layers can be sufficient enough to fit the training dataset, but with multiple layers, we can use fewer units per layer. But using more layers, the model becomes difficult to optimize. Thus, to find the best suitable architectural design, we must monitor the validation set error.

By the universal approximation theorem (Cybenko, 1989; Hornik, 1991): Given a feed-forward network with a linear output layer, at least one hidden layer, and an activation function as the sigmoid activation function, the model can approximate any Borel measurable function given enough hidden units. Thus, we don't need to build a very complex model to learn a non-linear function, since we are only considering continuous functions on closed and bounded subsets of $\mathbb{R}^n$. Even so, there could still be some issues. We could potentially have a learning algorithm that cannot learn the non-linear function. As described earlier, we could end up with a very high capacity which might overfit the model. There is also the possibility that the model cannot find the values of the desired parameters. Hence, we can not provide the model with a given function, just by examining the training set, which can be generalized to the test set. Thus, even though a model with only one layer can represent any function, the layer may be very large (including many units) and could have problems generalizing to a test set. Therefore, using a deeper model might be ideal since the layers would require fewer units.

Until now, we have mostly talked about the feedforward network. The forward pass feeds the input data to the neural network, then propagates up through the model until the final output $\hat{y}$ is reached. We will now introduce another supervised learning algorithm invented by Rumelhart et al. (1986) called **backpropagation**. The main goal of backpropagation is to minimize the loss function. To do so, the algorithm compares the errors between the predicted output with the actual output, to iteratively adjust the weights and biases in the neural network. In more detail, we calculate the gradient of the loss function wrt. the predicted outputs. This is then propagated back through the network. To do this, we need to apply the chain rule of calculus. This rule is applied when we have a composed function eg. $f(g(x))$. The derivative can, as just stated, be calculated using the chain rule of calculus hence $f'(g(x)) \cdot g'(x)$, or as $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$. This can then be generalized, by assuming that $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$ and that g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$ and that f maps from $\mathbb{R}^n$ to $\mathbb{R}$. Then we have

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i},$$

if $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$. If we then rewrite to vector notation we get

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \right)^\top \nabla_{\boldsymbol{y}} z.$$

Thus, we note that the gradient $\boldsymbol{x}$ can be found by $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \times \nabla_{\boldsymbol{y}} z$. When running the back-propagation, this is the calculation performed.

3.2.2 Regularization

According to Goodfellow et al. (2016), regularization strategies are strategies that are used for a machine learning problem. These strategies want to optimize the results from the test set hence minimizing the test error. Thus, using regularization strategies, the model should make better predictions on new inputs that the model has not been trained on. An important note is that regularization does not aim to minimize the training error. Throughout this section, we will cover two different strategies: Early stopping and dropout.[4]

When one uses a regularization strategy, the goal is to increase bias and lower variance hence, regularization strategies are in fact, regularization estimators. Optimally, a strategy that lowers the variance a lot, without increasing the bias too much, is considered a good strategy. Looking at Figure 3.3 from earlier, we can easily illustrate what the goal of regularization is. In the overfitting zone, we observe a decreasing bias and an increasing variance. As described above, we want to decrease variance on the cost of increasing the bias. Hence, using regularization we move from the overfitting zone to optimal capacity. This sounds easy but is not a simple task. Goodfellow et al. (2016, p. 229) describes it as: "*trying to fit a square peg, into a round hole*".

We have earlier discussed the advantages of having a validation set when training a large model. Here we tweak our model such that the model lowers the validation error. This is possible for some epochs, but eventually, the validation loss will converge and start to rise again even though the training loss is still decreasing. In this case, we want to introduce **early stopping**, which is an algorithm that returns the parameters that minimize the validation loss the most (Goodfellow et al., 2016). This method is highly effective and very easy to implement. Furthermore, the cost of using early stopping is also very low, since the algorithm just saves the parameters where the validation loss is lowest. To describe the algorithm, for every epoch where the validation loss is decreasing, the algorithm saves the parameters $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$ and saves the validation loss $v \leftarrow v'$. This will go on if $v > v'$. When the validation loss increases $v < v'$ the algorithm does not update $\boldsymbol{\theta}^*$. In this way, the algorithm saves the best parameters. This algorithm also returns the best number of training epochs since this is the number of epochs it requires to arrive at $\boldsymbol{\theta}^*$.

When training and testing models, there are several approaches one could use. One of the approaches is called **bagging** which is used to fit multiple models and average their prediction on the test examples. Bagging is possible when each model is not a large neural network. When having a large neural network, bagging would not be efficient since this would require very high memory and the running time would be slow (Goodfellow et al., 2016). To conquer this problem we introduce the concept of **dropout** (Srivastava et al., 2014). Using dropout, an inexpensive approximation can be made and thus, dropout can be used for large neural networks. We will now explain how the dropout algorithm works.

To visualize how dropout works, we examine Figure 3.5 by Goodfellow et al. (2016). This figure shows all combinations for a base network where a non-output is removed. This gives us a total of sixteen different subsets. These subsets can be obtained by removing one or multiple units from the base network. To do this, one could multiply the output value by zero.

[4] These two strategies are used in our Keras model which can be observed in Appendix 2.

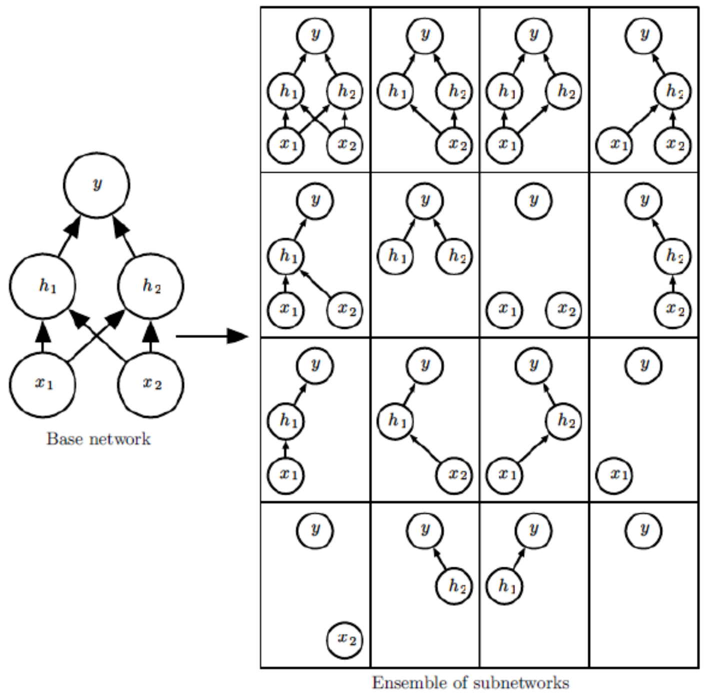


Figure 3.5: *Sixteen different sub-networks which can be obtained by removing a non-output unit from a given base network. In the given Base-network we have two input units* x_1, x_2*, two hidden units* h_1, h_2*, and an output unit* y*. Looking at the figure, only* $9/16$ *of the sub-networks consists of an input unit and a connected path from the input unit to the output unit. When the base network becomes wider, the fraction will become larger, and thus the problem becomes insignificant. Source: Figure 7.6 (Goodfellow et al., 2016).*

Thus, to train a model using dropout we consider the following minimization problem:

$$\min \mathbb{E}_{\boldsymbol{\mu}} J(\boldsymbol{\theta}, \boldsymbol{\mu}),$$

where $\boldsymbol{\mu}$ is a binary mask vector and chooses which units to drop, J is the cost function by the models' parameters $\boldsymbol{\theta}$ and $\boldsymbol{\mu}$. If we sample values of $\boldsymbol{\mu}$ we can get an estimate of its gradient. The dropout rate is given as a hyperparameter and determines the probability of each entry being 1. Hence, choosing a dropout rate equal to 0.5 corresponds to half of the inputs being ignored. Additionally, the hidden layers can have different probabilities assigned.

The dropout algorithm and the bagging algorithm are very much alike but still have some differences. When we have a bagging model, we train all models independently and the models will eventually converge to the given training set. This is not true for the dropout algorithm. When having a dropout, the models share parameters and thus when using the dropout algorithm, it is possible to have an exponential number of models. Furthermore, not all models have to be trained explicitly since this would lead to an untractable amount of memory. Here dropout only trains a tiny subset of the models, and shares the estimated parameters with the rest of the sub-network.

When running the dropout algorithm Srivastava et al. (2014) also showed that the computational cost was low. Furthermore, they argue that this algorithm does not significantly limit the desired model or how one is training the model. This can be trained with stochastic gradient descent and thus it is possible to use the dropout algorithm when modeling an RNN (Bayer and Osendorfer, 2014; Pascanu et al., 2013).

On a final note, when having dropout, we cannot totally ignore the computational cost. Even though this is true for a single model, this is not true for a complete system. When having dropout, one would experience that the optimal validation set error is lower than when not having dropout. In general, this is very good, but it will require a much larger model and more epochs to generate the parameters. Thus, in some cases, the latter cannot be ignored and hence, it could be better to exclude the dropout algorithm.

3.2.3 Optimizing Deep Models

There are many ways, in which optimization happens for deep learning models (Goodfellow et al., 2016). Writing proofs, designing algorithms, or something third. The most difficult form of optimization for deep learning models, however, is training the neural network. Therefore, it is often seen, that a huge amount of time on many different machines is used to solve an instance of this problem. Because of this, different techniques have been developed to help solve it. This is what we will be focusing on in this section. Specifically, we will focus on finding some parameters, $\boldsymbol{\theta}$, that will reduce a cost function $J(\boldsymbol{\theta})$.

Firstly, we discuss how learning is different from pure optimization (Goodfellow et al., 2016). A key factor here is that machine learning acts indirectly. This means that with machine learning, we care about some performance measure associated with the test set. This measure is then optimized indirectly, since what we actually do is reduce a cost function, and hope that the performance measure improves by doing so. Conversely, in pure optimization, the goal in itself is minimizing the cost function. We remember, that the cost function can be represented by

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y),$$

where L is a per-example loss function. Thus the cost function is an average over the training set. $f(\boldsymbol{x};\boldsymbol{\theta})$ is what we predict to output when the input is $\boldsymbol{x}$. $\hat{p}_{\text{data}}$ is the distribution, and y is the target output. The above cost function is defined over the training set alone. It is preferable to minimize the same objective function but over the data-generating distribution denoted by

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y). \tag{4}$$

The general goal of machine learning is to minimize this expectation also known as the risk. By doing so we minimize the expected generalization error and hence optimize the model. Looking closer at the expectation, we note $p_{data}(\boldsymbol{x}, y)$ as the true distribution. If this was given, we would not have a machine-learning problem but a normal optimization problem. Since this is unknown, the task is to determine the distribution during the training phase using the training set. If both the training and test set have a close lying distribution, this method would minimize the generalization error and thus the risk. But when we replace the true distribution with the empirical distribution $\hat{p}_{data}(\boldsymbol{x}, y)$ we are not minimizing the risk but the **emperical risk**. This can be described in the following way where m denotes the amount of data points,

$$\mathbb{E}_{\boldsymbol{x},y\sim\hat{p}_{data}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] = \frac{1}{m}\sum_{i=1}^{m} L\left(f\left(\boldsymbol{x}^{(i)};\boldsymbol{\theta}\right), y^{(i)}\right).$$

Several challenges arise from this "simple" method since this could lead to overfitting. If we increase the model's capacity too much, it would fit the training data perfectly since it would memorize it instead of predicting it. This means that this approach could be infeasible. Therefore, modifications need to be applied, and hence empirical risk optimization is rarely used. One very commonly used modification is to use a surrogate loss function.[5] This is simply a loss function that can be used to approximate the real loss but can sometimes perform better when optimizing classification errors. Another modification could be to use an **early stopping** algorithm.[6] This also differs from general optimization since we select the model which is selected using the algorithm. Therefore we do not halt at the local minimum but instead at an optimum found using a validation set.

Another thing that separates machine learning methods from pure optimization algorithms, is that the objective function in machine learning methods is often comprised of a sum of the training examples (Goodfellow et al., 2016). As an example, we can take a look at a maximum likelihood estimation problem. When viewed in log-space, this problem decomposes to a sum over each example

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=0}^{m} \log p_{\text{model}}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}).$$

And then, maximizing this sum is the same as maximizing the expectation over the distri-

[5] We do not use this in our code. Instead, we have experimented with different kinds of loss functions to see if there were any which performed better than others.

[6] From section 3.2.2 Regularization,

bution formed by the training set

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{x}, y; \boldsymbol{\theta}).$$

One of the most used properties of the objective function used in many optimization algorithms is the gradient

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{\text{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\text{model}}(\boldsymbol{x}, y; \boldsymbol{\theta}).$$

However, we know that calculating this expectation is very expensive since it requires us to evaluate the model on every example in the dataset. This can be avoided by taking random samples of the dataset and then calculating the expectations by taking the average over the samples, exactly as the mini-batches we introduced in section 3.1.6.

We remember, that the standard error of the mean introduced in equation (3) is given by $\sigma/\sqrt{n}$. Since we have $\sqrt{n}$ in the denominator, we note that it is not always worth it to use more examples to estimate the gradient. Consider two estimates of the gradient: One with 100 examples and one with 10,000 examples. Number two uses 100 times more computing power than number one, but only decreases the standard error by a factor of 10. In general, many optimization algorithms converge faster if they can rapidly compute approximations of the gradient instead of slowly computing the actual gradient.

According to Goodfellow et al. (2016), the optimization algorithms that use the entire training set are called batch gradient methods, while the optimization algorithms that only use one single example at a time are called stochastic methods. Algorithms used in deep learning usually fall between these two terms, using more than one but less than all the training examples. A good example of this is the stochastic gradient descent method, already described in this thesis.

There are multiple factors that contribute to determining the size of mini-batches. As shown above, larger batches will give a more accurate estimate of the gradient, however, the returns are less than linear. Additionally, the amount of memory needed scales with batch size. This might result in limitations from the hardware side of things. Following this, some hardware gets better runtimes with some specific sizes of arrays. This is why batch size is often seen in powers of 2; 16, 32, 64, and so on. Lastly, we note that small batch sizes can offer a regularizing effect as seen in Wilson and Martinez (2003). This is most prominent with a batch size of 1, which might require a lower learning rate since there will be a bigger variance in the estimate of the gradient. Therefore, the runtime might be quite high.

As discussed earlier, it is important to be able to compute unbiased estimates. To compute an unbiased estimate of the gradient from some samples, we require that the samples are independent. Since it is likely that the dataset is set up in such a way that subsequent data is somehow correlated to prior data, it is important that mini-batches are chosen randomly. Say one has a dataset of stock log returns, prices, or some other financial data from a range of different companies. The dataset will likely be structured in such a way, that the financial data from company A will be listed, then the data from company B, then C, and so on. If we draw examples from this list in order, the mini-batches would be very biased, since they would represent mainly one company out of all the companies in the dataset.

Returning to stochastic gradient descent, an interesting thing to note is that mini-batches on this optimization algorithm follow the gradient of the actual generalization error, seen in equation (4), if we make sure that none of the same examples are used more than once. As datasets increase in size more rapidly than computing power is able to follow, it is becoming more and more normal to only use each training example once, and sometimes to not go through the entire training set at all. An observation here is that when using a large enough training set, overfitting will not be an issue. Thus, the main concern will be underfitting and computational efficiency.

We have now explained how some machine learning techniques differ from general optimization and will now focus on some of the **challenges within optimizing deep models**. When using general optimization, one typically has a convex task that needs to be optimized. This can also be done for machine learning but when training neural networks, we must conquer the challenge of a non-convex task (Goodfellow et al., 2016). One of the most obvious challenges arises from the fact that non-convex sets can have multiple local minima.[7] Thus, when one finds a local minimum, we have no guarantee that this is also a global minimum whereas, for convex optimization, a local minimum is also a global minimum. This complicates things a bit but can be overcome. Looking at a model with latent variables, this might be non-identifiable since it could have weight space symmetry. This means that we could rearrange the hidden units in $n!^m$ ways. When analyzing these non-identifiable models, they have a very large number of local minima.[8] But, even though there are many, they are all equivalent which makes the challenge easier. The real challenge arises when the cost of the local minimum is much higher than the global minimum. This problem is very common and thus is a big problem because of the gradient-based learning algorithm which is used in neural networks. It is well-discussed whether this is a problem or not when training a large neural network. Dauphin et al. (2014) argues that the problem observed does not arise from the local minima itself, but from the locations of the saddle points.

A **saddle point**, also called a min-max point, is a point that is not a local minimum or maximum (Goodfellow et al., 2016). This can be seen in Figure 3.6, where the black dot illustrates exactly this point. All the gradients are zero in the orthogonal direction which is why it can be confused with a local minima or maxima. This occurs since the hession matrix has both positive and negative eigenvalues. If it was a local minimum, there would

[7] Neural networks often have many local minima.
[8] This could also be an uncountable infinitive amount of local minimums.

only be positive eigenvalues. Saddle points are very common in non-convex functions and

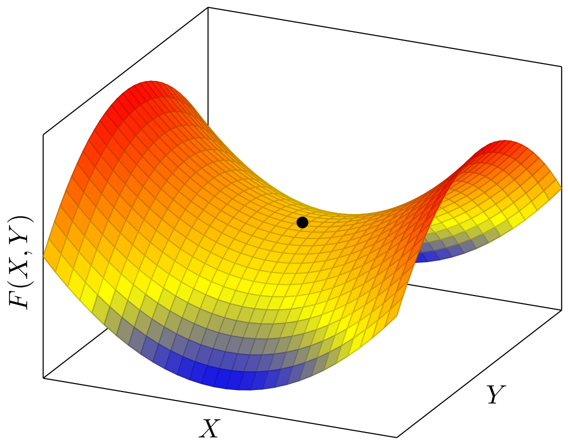


Figure 3.6: *Illustration of a saddle point. Source: Own work.*

even more common than local minimums. When examining spaces with high dimensions, there is almost no local minimums/ maximums but a lot of saddle points. But, when we are at a saddle point with a low cost, the Hessian matrix is more likely to be positive than negative due to the properties of many random functions. Thus, local minimums would have a low cost whereas local maximums would have a high cost. This general idea also applies to neural networks with nonlinearity (Baldi and Hornik, 1989) and thus this should not be a big problem.

Another potential challenge could be the cliff challenge which leads the gradients to explode (Goodfellow et al., 2016). In Figure 3.7 we observe exactly this problem. The problem occurs when there are multiple layers since this generates very steep cliffs in the loss function. When making a gradient update step toward a cliff, this can cause the parameters to “jump” away from the cliff which results in loss in the optimization. This can both happen when making the gradient update step towards a cliff and off a cliff. This is very clearly illustrated in the figure, where we observe the problem to the left. Fortunately, there is a solution to this challenge, which is the use of the gradient clipping heuristic. This algorithm “activates” when the gradient descent algorithm wants to make a large step. The clipping gradient then reduces the step size and specifies in which direction the step should be taken. This is shown in the right picture where we don’t observe a jump anymore.
Another challenge in optimizing deep models is how some parameters should be initialized.

According to Goodfellow et al. (2016), for optimization algorithms, it is often the case that they are not iterative, and thus just solve for their objective. For others, they might be iterative, but arrive at a solution fast enough, regardless of how they are initialized. For deep learning training algorithms, this is not usually the case. Here it is often necessary for the user to specify some initial point to start from. Since deep learning is so complex, the

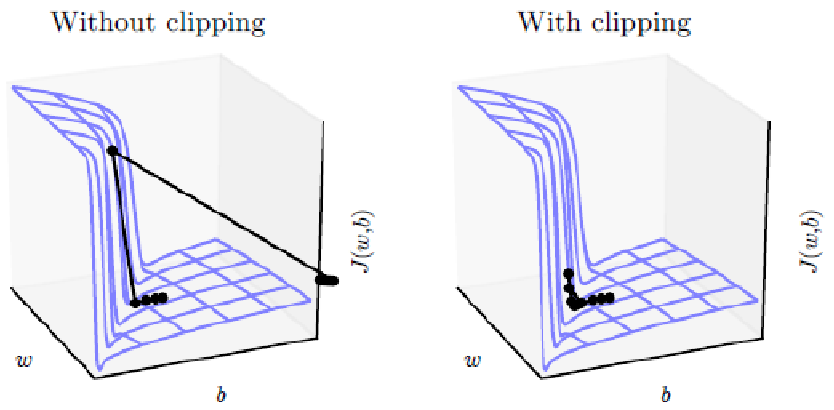


Figure 3.7: *A cliff can cause the gradient to explode. The left picture illustrated how a small gradient update step can make the parameters jump off the cliff. The left picture shows that this does not happen when using the gradient clipping heuristic algorithm. Source: Figure 10.17 (Goodfellow et al., 2016).*

initialization point is often crucial to the results obtained as well. Sometimes, the initial point might result in the model not converging, while at other times it might result in how quickly the model converges. As neural network optimization is not so well understood in the literature yet, designing initialization strategies is not an easy task. These strategies are often only based on getting nice properties when the model is initialized, but it is not known which of these properties are kept and why they are kept, when learning has begun. Another problem is that some initial points are helpful in optimization, but not in generalization.

One thing about initialization is known, however. This is, that initialization parameters have to "break symmetry" between layers. If we have two connected hidden layers with the same activation function and the same inputs, then these two layers must be initialized differently. If they have the same initialization parameters, then these units will constantly be updated in the same way. Avoiding this helps make sure that no input patterns and gradient patterns are lost.

Biases for a model will typically be chosen as constants, while the weights will be initialized randomly. Commonly they are drawn from a Gaussian or a uniform distribution. How this initial distribution is scaled has a big effect on how the network generalizes and how the model is optimized. Weights initialized with large values will be better at breaking the symmetry, described above. If the initial weights are too large, however, it may result in exploding values and extremely high sensitivity to even the smallest changes in the input. This problem can be somewhat resolved with gradient clipping as described above. Large values can however also result in the activation function becoming saturated and the gradient completely disappearing. All these factors contribute to determining the best initial scale of the weights.

While the approach to setting the weights might seem complicated, the approach to setting the biases is simpler, though it should be coordinated with how the weights are set. Often, it is fine to initialize biases with 0's, but there are some situations where non-zero values are beneficial:

- If the bias is for an output layer, it can be helpful to initialize the bias to match the marginal statistic for the output. That is, initialize the bias to match the marginal distribution of the input. This is typically done in models, where the output should resemble the input.
- If using the ReLU activation function, it can be helpful to choose a bias that does not cause too much saturation at the initial point. An example of this could be setting the bias of a hidden ReLU layer to 0.1 instead of 0.

It is quite clear, that there is no clear-cut way to initialize the parameters of a deep model optimizer. An interesting idea is to use machine learning for this task. A good strategy may be to initialize a supervised model with some parameters learned from an unsupervised model trained on the same inputs.

3.3 Models

The goal of this model section is to explain how different models can process sequential data. The idea is to input a lot of training data and output a prediction on the stock log return. The following models have different approaches and thereby slightly different results. Overall, the advantage of using recurrent networks is the models' ability to share parameters. Hence, it is possible to extend and apply the recurrent network to data that does not have the same length.

In general, these models use computational graphs to formalize the computational structure (Goodfellow et al., 2016). In the following models, we are only considering recurrent models which can be described as the following dynamic system, where it is driven by an external signal $\boldsymbol{x}^{(t)}$

$$\boldsymbol{s}^{(t)} = f\left(\boldsymbol{s}^{(t-1)}; \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right).$$

From this, we can define the hidden unit in an RNN as

$$\boldsymbol{h}^{(t)} = f\left(\boldsymbol{h}^{(t-1)}; \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right). \tag{5}$$

3.3.1 Recurrent Neural Network (RNN)

Recurrent Neural Network (RNN) is a type of neural network that can be used to analyze sequential data. This type of analysis can be used to predict a variety of things, among which, stock log returns are one of them.

Accoridng to Goodfellow et al. (2016), the model takes an input vector x_t which consists of some kind of financial factors e.g. dividend to-price ratio (DTOP) and so on. Now using multi-layer perception, the RNN model tries to predict e.g. the stock return given the factor inputs. The model calculates the hidden unit by a set of weights W and a hyperbolic tangent (non-linear function) function which returns values between -1 and 1. Hence,

$$h = \tanh\left[W \cdot x + b\right], \tag{6}$$

where b is the bias. The Neural Network will "update" the hidden unit every timestep, such that $\mathbf{h^{(t)}}$ becomes a summary of task-relevant aspects of the previous input data. It is important to notice that the hidden unit will "forget" some of the aspects of the data. We can represent a single RNN module as in figure 3.8. We can formulate the problem using equation 5, hence

$$\begin{aligned}\boldsymbol{h^{(t)}} &= f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) \\ &= g^{(t)}\left(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, ..., \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right).\end{aligned}$$

From this, we clearly observe that the sequence length does not matter since the learned model will always have the same size. A further advantage is that we can use the same function f with the same parameters at every timepoint (t). Looking at the RNN module

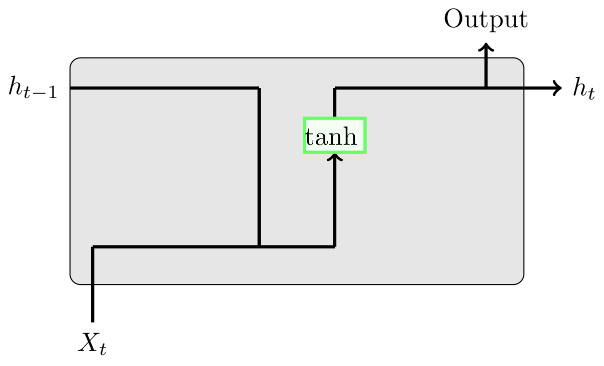


Figure 3.8: *A single RNN module. Source: Own work.*

in figure 3.8 we observe it takes two inputs, namely the data X_t and the hidden vector h_{t-1}. The data x and the hidden vector h are then run through the neural network and the h_t

is computed as shown in equation 6. There are multiple design choices when creating an RNN. In the shown module, we observe an output (outputs h_t). In some cases, it could be preferred to return an output for each timestep. Using our stock return approach, we would only want to return the output at the very last RNN module. This can be modeled as in figure 3.9. The hidden vector h will in the end be dependent on previously hidden vectors h together with previous input data x, hence this will provide us with a prediction that depends on all the data. Furthermore, every time we move on to a new RNN module the bias will change since this will learn the behavior of the data inputs.

$$\begin{tikzcd}
\boxed{\text{RNN(1)}} \arrow[r, "h_1"] & \boxed{\text{RNN(2)}} \arrow[r, "h_2"] & \cdots \arrow[r, "h_{x-1}"] & \boxed{\text{RNN(x)}} \arrow[r, "h_x"] & \text{Final output} \\
\text{Input 1} \arrow[u] & \text{Input 2} \arrow[u] & & \text{Input x} \arrow[u] &
\end{tikzcd}$$

Figure 3.9: *Illustration of a Recurrent Neural Network. Source: Own work.*

When running the RNN, the input data is concatenated with the hidden vector as shown in Figure 3.8. This happens in all our RNN modules until we have inputted all the data and hence, we are in the last RNN module. In this way, we end up with a model which uses all previous data to make a prediction.

There are also some limitations to using the RNN model. When using an RNN, it keeps track of all the data is has been given as input. Even though this might sound good, it will not always be. An example of this is easily explained in the application “natural language processing”: We want the RNN to guess the next word in a speech from Obama. In the first sentence, he mentions his wife Michelle. He then goes on for 5 minutes about a lot of other things where he also mentions his daughters’ names. In the end, we want to predict the following: “The name of Obama’s wife is ______”. Using RNN, a possible outcome could look as the following:

$$\text{output} = \begin{cases} \text{Michelle} & \text{prob} = 0.33, \\ \text{Malia} & \text{prob} = 0.33, \\ \text{Sasha} & \text{prob} = 0.33. \end{cases}$$

Here we clearly see that while an RNN model is a fine prediction tool, it does not remember the context of what we are trying to predict. This is due to the fact that The RNN does not have the capacity to “remember” for longer prediction tasks. Therefore, optimally we would need some kind of model that is capable of remembering certain aspects of the data and forgetting other aspects that are no longer relevant.

3.3.2 Long Short-term Memory (LSTM)

This section explains the idea of an LSTM model. This type of model will not be used on our data. Even though, it is still a very relevant model to investigate when considering sequential data.

According to Goodfellow et al. (2016), the Long Short-term Memory (LSTM) structure is almost the same as in the RNN. The difference here is that the model introduces the aspect of long-term memory. This makes it more capable of learning long-term dependencies. To submit this aspect to the model, all we have to do is introduce a memory cell. In the visualization below we see exactly this; the only difference from the overall RNN structure is that we have now introduced a memory cell, which we pass forward to the next module throughout the model. The LSTM consists of a total of four activation functions. Three of

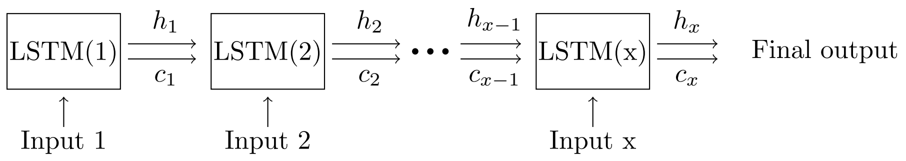


Figure 3.10: *Illustration of a Long Short-term Memory structure. Source: Own work.*

them are through a sigmoid function that returns values between 0 and 1, and one through a tanh function that returns values between -1 and 1. The intuition behind this is rather simple. Sigmoid functions are typically used to decide what information is important enough to be held onto and what information should be dropped. Tanh on the other hand is used to keep the values in check. Through the network, some values will undergo many operations, and might become so large, that other information might seem insignificant. To combat this, we can use the tanh-function to push all values to be between -1 and 1.

All the different activation functions use different weights and different biases. We will go through all the networks in more detail shortly. First, A single LSTM module is illustrated in figure 3.11. Here we notice that the module is a bit more complicated than the RNN counterpart. We have the same input but now add the memory cell from the previous module as well. This is what is keeping track of the long-term memory. Additionally, we now also output this memory cell to be used in the next module.

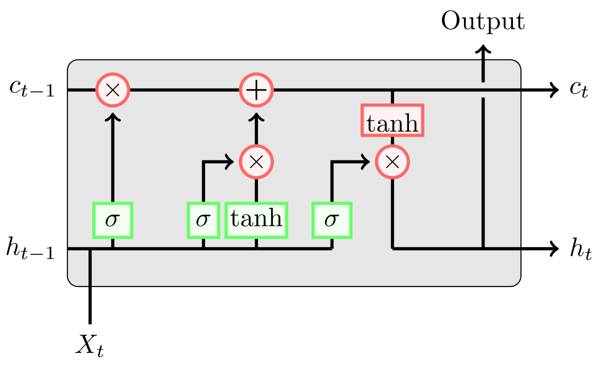
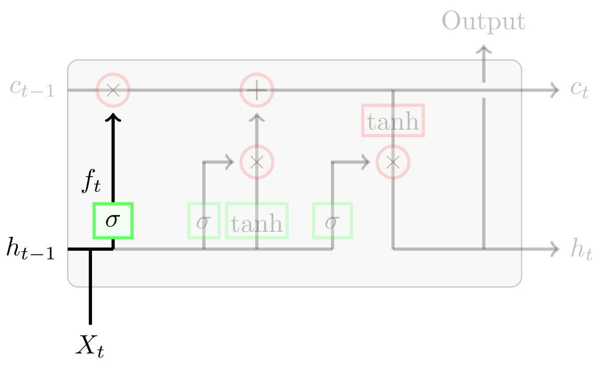


Figure 3.11: *A single LSTM module. Source: Own Work.*

As mentioned before, we will now go more in depth with the LSTM module. One can say, that LSTM works in four steps:

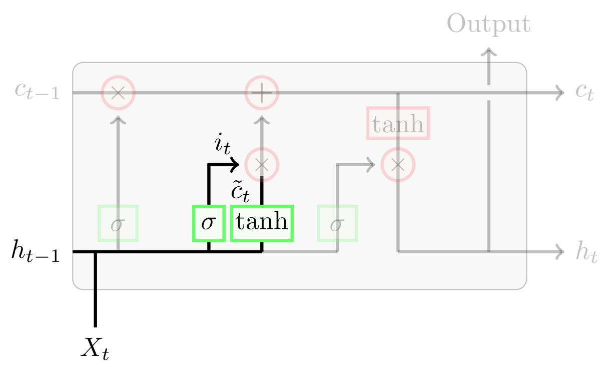
First, we want to decide which information from the memory cell we should forget. This is done in the *forget* gate, which uses the first of the four activation functions contained in LSTM. The forget gate is pictured Figure 3.12: This gate takes the input and concatenates



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

Figure 3.12: *LSTM illustration walkthrough. Source: Own work.*

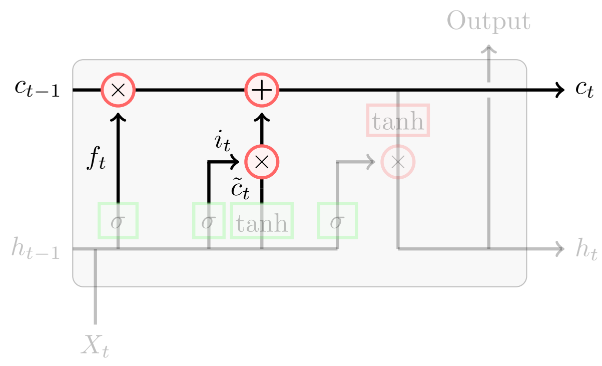
this with the hidden vector from the previous module. This is then multiplied with the weight matrix, W_f, and then a bias is added. All of this is passed through a sigmoid function that returns values between 0 and 1.
Next up is the decision on what new information is going to be stored in the memory cell. This is done in two parts, and is pictured in Figure 3.13:



$$i_t = \sigma(W_i \cdot [h_{t-1}, X_t] + b_i)$$
$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, X_t] + b_c)$$

Figure 3.13: *LSTM illustration walkthrough. Source: Own work.*

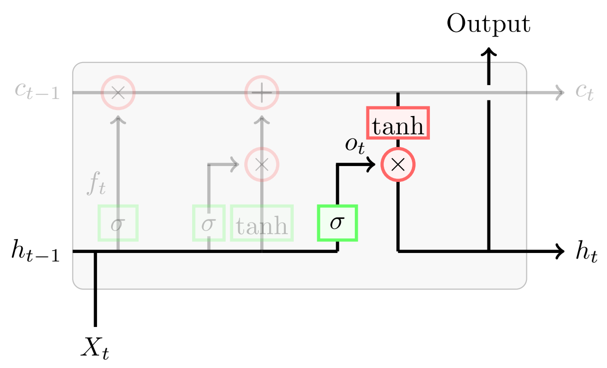
Firstly, the *input* gate is used. This works in the same way as the forget gate, just using a different weight matrix, W_i, and bias. Again, this is passed through a sigmoid function. Next, we need to create a vector, $\tilde{c}_t$, of new values that could be added to the memory cell. This is calculated in the same way as the forget and input gates, just again using a different weight matrix, W_c, and bias. This is passed through the tanh function, which outputs values between -1 and 1.
Now we can update the memory cell. This is pictured in Figure 3.14: Firstly, we pointwise



$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$

Figure 3.14: *LSTM illustration walkthrough. Source: Own work.*

multiply the old memory cell, c_{t-1}, by the output of the forget gate, f_t. Since f_t consists of values between 0 and 1, we can see values close to 0 as something to forget and values close to 1 as something to keep. Next, we add the new candidate values, which are scaled by how much we decided to update the memory cell ($\tilde{c}_t$ being between -1 and 1).
Finally, we can decide what is outputted. This is pictured in Figure 3.15: The final gate is the *output* gate. The output is based on the memory cell but will be an altered version. First, we run the input and the hidden vector from the previous cell through a sigmoid



$$o_t = \sigma(W_o \cdot [h_{t-1}, X_t] + b_o)$$
$$h_t = o_t * \tanh(c_t)$$

Figure 3.15: *LSTM illustration walkthrough. Source: Own work.*

function, to decide what part of the memory cell we are going to output. This works in the same way as the previous gates, but again, we use a different weight matrix, W_o, and bias. The memory cell is then pushed through the tanh-function to get values between -1 and 1. Now, we can pointwise multiply the two, to get the desired output.

Now we have a model that is much more suited to tackle the problem of long-term dependencies. Take our "natural language processing" example from the previous section. Now using our LSTM model on Obama's speech, the name of Obama's wife, Michelle, is stored in the memory cell as relevant information. Thus, when doing our prediction: "The name of Obama's wife is ______", the model has a much easier task of choosing the correct answer:

$$\text{output} = \begin{cases} \textbf{Michelle} & \textbf{prob} = \mathbf{0.90}, \\ \text{Malia} & \text{prob} = 0.05, \\ \text{Sasha} & \text{prob} = 0.05. \end{cases}$$

4 Data collection

In this section, we go into great detail about everything related to the data collection. Since we want a high-quality dataset, we are not able to download this directly from any website. Therefore a lot of manipulation is required to obtain the dataset that not only is high quality, but also fits our neural network approach.

Firstly, we go through the implementation details. Here, we highlight the software used in this thesis as well as the computing environment used. We also go through the process of downloading the data from the platform WRDS. Next, we go through everything related to the fundamental factors in our dataset. We show the fundamentals downloaded to compute the factors as well as how every factor is computed. We also go through the macroeconomic factors used; where they are downloaded, how we manipulate the data for them, and how they develop over time. After this, we examine the final steps needed to have a finished dataset. This includes looking at the correlation between all factors. Furthermore, other issues with the data are dealt with. Lastly, we go through the data pre-processing involved with using a dataset in an RNN model. This involves making some manipulations and setting up the data so the input size fits the model we intend to use. We also define our training and test sets used.

4.1 Implementation Details

Software. We have used several different software throughout this thesis. All data manipulation and model implementation has been done using Python 3 (Van Rossum and Drake, 2009). Within Python, we have used several different libraries. The library numpy (Harris et al., 2020) is used for many basic array operations etc. The library Pandas (pandas development team, 2020) has been used for data manipulations and data visualization. The library SciKit Learn (Pedregosa et al., 2011) has been used for MinMax scaling and MSE calculations. The library TensorFlow (Martín Abadi et al., 2015) has been used for creating the model. Within this matter, the library Keras (Chollet et al., 2015) has been used for creating the model. Finally, the library Matplotlib (Hunter, 2007) has been used for visualizing several different figures, etc.

Computing environment. All of the above was implemented on a computer with 8GB RAM with Intel i7 CPU 1.8 GHZ processor and a computer with 16GB RAM with AMD Ryzen 5 six-core CPU 3.6 GHZ processor.

Data. Since the purpose of this thesis is to predict the monthly stock log returns of a collection of specific US firms, we create our own dataset. The reason for us creating our own dataset instead of using some existing dataset is that this gives us some freedom in choosing exactly what type of firms and what type of factors we want to include. As we will see, our dataset ends up being very unique and very large. As is often the case in situations like these, the creation of such a dataset comes with some issues, that we will highlight later in this section.

The data for our thesis consists of many different aspects, that affect the stock log returns of the companies included. Most prominently, we work with company factors that directly highlight the financial situation of the companies in the dataset. Additionally, we consider certain macroeconomic variables, that will hopefully help highlight overall trends in the market.

To create the above-mentioned factors, we need different company fundamentals. These are collected from the Wharton Research Data Services (WRDS) database (WRDS, n.d.). On WRDS there a many different types of databases. We collect data from "Compustat North America" which is a database of US and Canadian fundamental and market information on publicly held companies. Our thesis focuses on monthly stock log returns which makes this database a perfect fit for our work since it provides more than 100 quarterly and monthly income statement items, balance sheet items, and more, dating all the way back to 1962. To access this database we follow the path: WRDS / Get Data / Compustat - Capital IQ / Compustat / North America.

For the dataset, we have prepared a time series dataset, containing monthly and quarterly data starting from February 1, 1990, ranging to December 31, 2022. This provides a total of 395 months of data. Furthermore, the dataset contains all companies available from WRDS/ Compustat - Capital IQ, that are listed on three exchanges. These are the New York Stock Exchange ("NYSE", 2023), the American Stock Exchange (Now NYSE) ("NYSE", 2023), and the NASDAQ-NMS Stock Market("NASDAQ", 2023). On these three exchanges, we are able to collect data from a total of 19,881 companies, however, since we are naturally only interested in active companies, we drop all inactive companies. This provides a total of 6406 active companies, that have all been listed on either of the three stock exchanges between 1990-2022. On average our dataset contains 2925 unique active firms per year, but interestingly, the number of unique firms present in the dataset is steadily increasing each year, as seen in figure 4.1. The figure also shows how the distribution of unique firms each year would look if inactive firms were not omitted.
The number of unique firms each year is, as stated above, increasing. This is obviously because of the fact, that inactive firms are dropped. If this was not the case, the number of firms would be more consistent.

4.2 Factors

For each company, we download a total of 14 quarterly fundamentals and 5 monthly fundamentals, that we use to calculate our factors. These fundamentals are shown in Table 4.1. After downloading the before-mentioned fundamentals, we have computed a total of 26 fac-

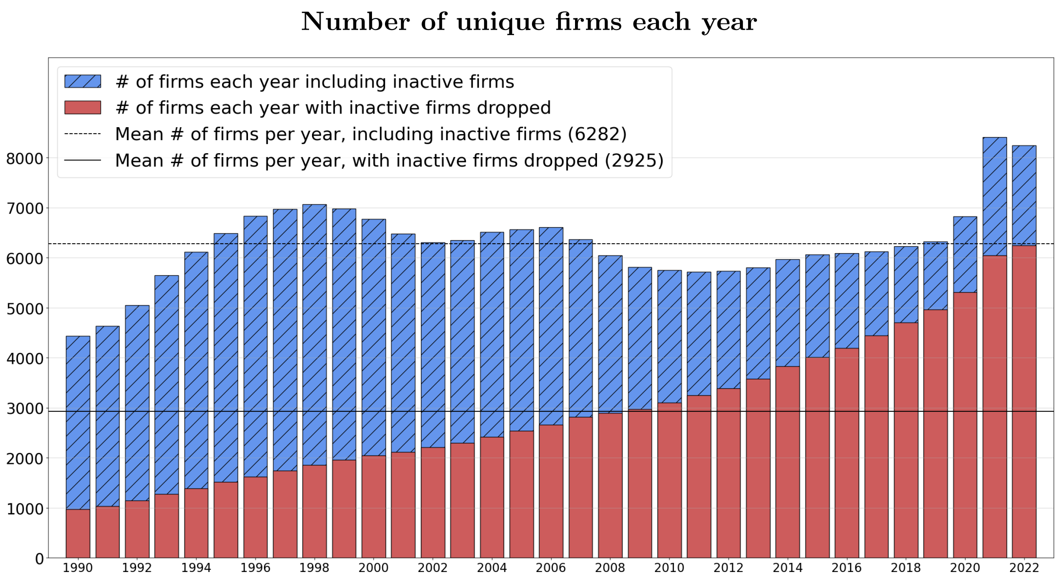


Figure 4.1: *The number of unique firms in the dataset for each year. Red solid bars represent the number when inactive firms are dropped, while the blue shaded bars represent the number when including inactive firms. The mean number of firms is also shown (value in legend). Source: Own work.*

tors, all divided into different categories, to get a well-rounded view of the specific company and what drives its stock log returns. We have listed the computed factors in Table 4.2. The categories and the belonging factors (with number of factors in each category in parentheses) are as follows:

- **Momentum (5).** This category describes the tendency of winning stocks to continue performing well. Here we include both momentum factors and reversal factors, that show reversals in the companies' stock prices, that is if the overall price direction changes. We have a number of factors in this category; SHOREV, LONREV, 1M MOM, 1Y MOM, and MED MOM.

  SHOREV is defined as the previous month's return, LONREV is defined as the return from month (t-60) to (t-13), while MED MOM is defined as the return from month (t-12) to (t-2). The 1M MOM and 1Y MOM factors are calculated using the following formula

$$\text{Momentum} = V - Vx,$$

  where V is the latest price and Vx is the price x months ago. Medhat and Schmeling (2021) examine short-term reversal and short-term momentum, and find that these may suggest some information on returns. Kelly et al. (2021) construct a factor pric-

Fundamental	Description	**Fundamental**	Description
ACTQ	Current Assets	LCTQ	Current Liabilities
ATQ	Assets Total	LTQ	Liabilities Total
CAPXY	Capital Expenditures	PRCCQ	Price Close
CHQ	Cash	SALEQ	Sale Turnover
COGSQ	Cost Of Goods Sold		
CSHOQ	Common Shares Outstanding	CSHOM*	Shares outstanding - Monthly
DLCQ	Debt in Current Liabilities	CSHTRM*	Trading volume - Monthly
DLTTQ	Long-Term debt	DVPSXM*	Dividends per share - Ex date - Monthly
DPQ	Depreciation and Amortization	PRCCM*	Price close - Monthly
EPSPXQ	Earnings Per Share	TRT1M*	Monthly total return

Table 4.1: *All quarterly and monthly (denoted by *) fundamentals downloaded (WRDS, n.d.). Source: Own work.*

Factor	Description	**Factor**	Description
STOP	Sales-to-price ratio	BTOP	Book-to-price ratio
DTOP*	Dividend-to-price Ratio	ABS	Accruals-balance sheet version
VSAL	Variability in sales	ANN VOL*	Annualized volatility
SGRO	Historical sales per share growth rate	EGRO	Historical earnings per share growth rate
STOM*	Monthly share turnover	DTOA	Debt-to-Assets Ratio
STOQ*	Quarterly share turnover	AGRO	Asset growth
IGRO	Issuance growth	CXGRO	Capital expenditures growth
CX	Capital expenditures	ATO	Assets turnover
GP	Gross profitability	GM	Gross margin
MAD*	Maximum drawdown	SEASON*	Seasonality
LNCAP*	Log of market capitalization	SHOREV*	Short-term reversal
1M MOM*	1 month momentum	LONREV*	Long-term reversal
1Y MOM*	1 year reversal		
MED MOM*	Medium momentum		

Table 4.2: *Computed quarterly and monthly (denoted by *) factors. Source: Own work.*

ing model using long-term reversal and one-year momentum and find that these can be used to generate strong return predictions. Based on these contributions to the literature, we include these factors in our dataset.

Dixon and Polson (2020) Consider a wide array of factors in their deep fundamental factor model. Alongside this paper, we also consider factors from the MSCI FaCS MSCI (2020), which is a Factor classification standard, designed to provide investors with the tools to implement a factor investing strategy, and in our case, a factor model. The following categories we have thus included since they appear in the above-mentioned sources.

- **Dividend Yield (1).** Here, we highlight how a company's dividend yield impacts its stock price and return. Again, there is only one factor in this category; DTOP.

  This factor is calculated by dividing the trailing 12-month dividend per share by the current price.

- **Earnings Quality (2).** By earnings quality, we can measure how reliable the company-reported earnings are in determining the company's current performance and predicting future performance. Here, we consider the factors; ABS, and VSAL.

ABS is calculated using

$$\text{ABS} = [(\Delta CA - \Delta Cash) - (\Delta CL - \Delta SDT) - Dep]/TA,$$

where CA is current assets, CL is current liabilities, STD is short-term debt, TA is total assets, and Dep is depreciation.

VSAL is the variability in sales and is thus computed as the standard deviation of company-reported sales over the last five fiscal years.

- **Growth (2).** This category regresses the company's historical growth, to find growth rates for earnings and sales. This gives some idea of how the return might evolve with time. Here we consider the two factors; EGRO, and SGRO.

  Both factors are calculated in a similar way. For EGRO we use earnings per share, which is regressed against time over the last five fiscal years which gives a slope coefficient that we can then divide by the average annual earnings per share. For SGRO we do the same, but instead of earnings per share, we use reported sales.

- **Leverage (1).** We also need to consider the leverage of the companies. This is done here by the factor; DTOA.

  DTOA is the debt-to-asset ratio and is calculated using

  $$\text{DTOA} = TD/TA,$$

  where TD is total debt and TA is total assets.

- **Liquidity (2).** We consider the liquidity of the companies through this category, since this gives a highlight into the financial health of the companies. Here we consider STOM and STOQ.

  STOM is computed by

  $$\text{STOM} = \ln\left(\frac{V}{S}\right),$$

  where V is trading volume for the current month and S is shares outstanding. When we have STOM, we can calculate STOQ by

  $$\text{STOQ} = \ln\left[\frac{1}{T}\sum_{t=1}^{T}\exp\left(STOM_t\right)\right].$$

Above, T is set to 3, and $STOM_t$ is the STOM factor for the current month, t.

- **Profitability (3).** Looking at profitability, which is a measure of a company's profit relative to its expenses, we can get an insight into how well the company is doing, and by extension: how the return for the company's stock is doing. Here we look at a number of factors; ATO, GP, and GM.

  These three factors are computed with

$$\begin{aligned}\text{ATO} &= Sales/TA,\\ \text{GP} &= (Sales - COGS)/TA,\\ \text{GM} &= (Sales - COGS)/Sales.\end{aligned}$$

  Above, $Sales$ is the most recently reported company sales, TA is total assets, and $COGS$ is the most recently reported cost of goods sold.

- **Size (1)**
  It might also be helpful to consider the size of the companies. For this, we consider; LNCAP.

  This factor is simply computed as the natural logarithm of the market cap of the firm.

- **Value (2)**
  We also consider the category "value". These are certain factors that highlight the value of the companies considered. Here we use; BTOP, STOP.

  Here, BTOP is calculated by taking the last reported book value of common equity and dividing it by the current market cap. STOP is the last reported sales divided by the current market cap.

- **Volatility (1)**
  We consider volatility as a category, to hopefully be able to capture the volatile tendencies of stock log returns. For this, we consider the factor ANN VOL.

  To calculate ANN VOL, We take the standard deviation over the past year's monthly returns, and multiply this with $\sqrt{12}$.

- **Management quality (4)**
  Additionally, we look at the management quality of the companies as well. Here, we consider four factors; AGRO, IGRO, CXGRO, and CX.

  Here we again use regression. For AGRO we take the total amount of company assets and regress against time over the past five fiscal years. Then we take the slope coefficient and divide it by the average amount of assets. Cooper et al. (2008) examine asset growth and find that this is a strong predictor for future stock returns.

For IGRO we take the total number of shares outstanding and regress against time over the past five fiscal years. Then we take the slope coefficient and divide it by the average number of shares outstanding. Greenwood and Hanson (2012) show that characteristics of stock issuing firms can be used to forecast important factors in stocks' returns.

For CXGRO we take the total reported capital expenditures and regress against time over the past five fiscal years. Then we take the slope coefficient and divide it by the average amount of capital expenditures.

For CX we take the most recent capital expenditures and scale them by the average of capital expenditures over the last five fiscal years.

- **Prospect (1)**
  We use prospect to give an insight into what one can expect in terms of returns from the stock. Here we consider the factor MAD.

  To calculate this, we consider the difference in the stocks' highest and lowest price in the last 12 months, and is thus a measure of the maximum amount an investor could lose if they purchased and sold the stock in the specific period.

- **Seasonality (1)**
  Seasonality is a specific trait in time series data, where a regular and predictable change occurs in the data around the same time every calendar year. This we can use to assess whether the return behaves in specific patterns at certain times of the year. Here we consider the factor; SEASON.

  For seasonality, we look five years back and find the average monthly returns over this period. Additionally, we find the "seasonal" average return during the same five-year period. These averages are then divided to find the seasonality factor. More explicitly, say we are standing in June 2022 and want to calculate the seasonality factor for a specific stock for this month. We would then find the "total average" by averaging the returns from the last 60 months. Secondly, we would find the average returns of the past five Junes and then finally divide the seasonal average by the total average.

Macroeconomic Factors

As mentioned previously, we also include certain macroeconomic factors in our dataset. We do this to hopefully highlight certain trends in the market. Additionally, macroeconomic factors will be helpful in periods with mainly negative returns (Chen, 2009). The macroeconomic variables are found on the Federal Reserve Economic Data website of the Federal Reserve Bank of St. Louis. Here we find a number of macroeconomic factors listed below:

- **Federal funds interest rate**. This is found on "Federal Reserve Economic Data - FEDFUNDS" (2023). This factor refers to the interest rate that banks charge other institutions when lending cash from the bank reserves overnight. This means, that borrowing becomes either more or less expensive, and will thus have an impact on the economy and the financial market (Maio, 2014).
- **GDP growth rate** found on "Federal Reserve Economic Data - GDP" (2023). We include this factor to better highlight the financial market. Lower GDP growth rates will most likely indicate lower spending and optimism as one may observe in a bear market, while higher GDP growth rates will indicate the opposite. Thus, a model with information about the GDP growth rates will be more able to explain stock returns (Vassalou, 2003).
- **Three-month US treasury rate**, found on "Federal Reserve Economic Data - TB3MS" (2023). The three-month US treasury rate is the yield received when investing in a three-month government treasury security. This factor is important when looking at the overall US economy. When the rate goes up, the cost of capital goes up which can result in returns going down in the long run, and vice-versa, in the same vein as the FEDFUNDS factor.
- **Difference between 3-year treasury rates**, found on "Federal Reserve Economic Data - DGS3" (2023) and 10-year treasury rates found on "Federal Reserve Economic Data - DGS10" (2023). The 10-year is subtracted from the 3-year and added to the dataset. This is also called a yield spread, and is a difference between the quoted rate of return on different debt instruments. Changes in the yield spread can signal changes in the underlying economy, and thus we consider this factor for its ability to predict US recessions (Resnick and Shoesmith, 2002).
- **Unemployment rate** found on "Federal Reserve Economic Data - UNRATE" (2023). This factor will, again, be helpful in highlighting the situation in the financial market. Boyd et al. (2005) note, that an announcement of rising unemployment is good news for stocks during economic expansions and bad news during economic contractions. We hope to be able to capture this effect.
- **Volatility index VIX**, found on "Federal Reserve Economic Data - VIX" (2023). Again, we want to highlight the financial situation in the market, this time in terms of volatility, and thus we incorporate the VIX index. Rubbaniy et al. (2014) find that implied volatility is a good predictor of 20-day and 60-day forward-looking returns, and since we are trying to predict next month's log returns, this will help us.

For every macroeconomic factor monthly data is chosen (except the GDP growth rate, which is chosen in quarterly intervals) in the interval of February 1st, 1990 until the latest date available at the time of this thesis, December 31st, 2022. The development in the macro factors is shown in figure 4.2.

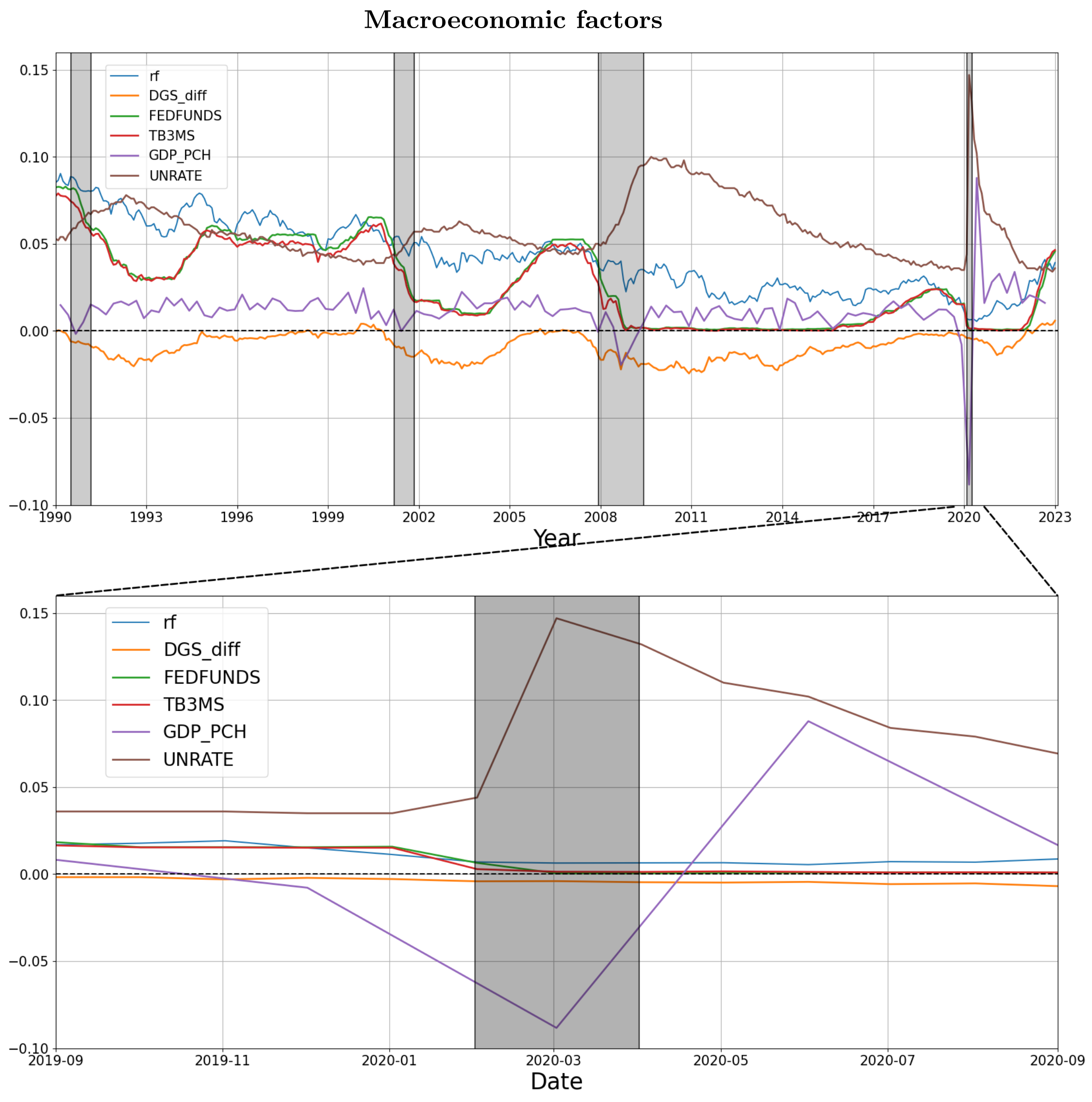


Figure 4.2: *Development in macro factors. Shaded areas represent major US. recessions gathered at "Federal Reserve Economic Data - Recession Bars" (2023). These are in order: "The early 1990s recession" July 1990-March 1991, "The early 2000s recession" March 2001-November 2001, "The Great Recession" December 2007-June 2009, and "The COVID-19 recession" February 2020-April 2020. The bottom plot shows the latter recession zoomed in, to better highlight what happens. Source: Own work.*

We see that these factors are able to help explain some trends in the economy. The shaded areas in the figure represent major US. recessions, where everything from interest rates to change in GDP drops, and unemployment rises. This is especially significant in the COVID-19 crisis highlighted in the bottom plot of figure 4.2, where unemployment rises to an all-time high of almost 15%, and change in GDP sits at an all-time low of around -9%.

Now, for the actual data, some manipulations are done. For the risk-free rate, the federal funds interest rate, the three-month US treasury rate, the difference between the 3-year and 10-year treasury rates, the unemployment rate, and VIX, the data frequency is as mentioned before monthly. Additionally, the data is customized in such a way that everything is divided by 100 (except for VIX, which is an index). This is done to be consistent with our dataset. for the GDP growth rate, the frequency is quarterly. Here the data is also customized. Units are changed to percent change as to show the growth rate, and thereafter the data is again divided by 100 to be consistent. Since macroeconomic variables are lagging, that is, the data represented on, say, the 1st of January in the dataset actually covers the month of December, we have to shift the data up by 1, so it is in the right overall position in our dataset. Finally, we can concatenate this macroeconomic dataset with our primary dataset described in the above section.

4.3 Data correlation

With the dataset now complete, we are able to explore it in further detail. In Figure 4.3, we are able to do just that. This figure shows how every factor in our entire dataset is correlated with each other. The idea here is, that we are interested in, whether our choice of factors is good, and that we will be able to predict log returns. It is well known in the literature, that a good factor set contains factors that are uncorrelated with each other Hall, 2000. Thus, it stands to reason that for our factors to be effective, they should be uncorrelated with each other. In general this is true in our dataset, where most factors have a very small correlation. We do have some factors that are slightly correlated, and some that are highly correlated as seen in Figure 4.3. Ideally, we do not want this and prefer that they are completely uncorrelated with each other. Obviously, every factor's correlation with itself is 1. There are however three other cases where we observe this strong correlation. These are:

- STOM with STOQ (monthly share turnover & quarterly share turnover, respectively.),
- GP with ATO (gross profitability & asset turnover, respectively),
- FEDFUNDS with TB3MS (federal funds interest rate & three-month treasury bill, respectively).

These factors being one-to-one correlated means, that they basically provide the model with the same information, and thus one of them is irrelevant. STOM and STOQ make sense

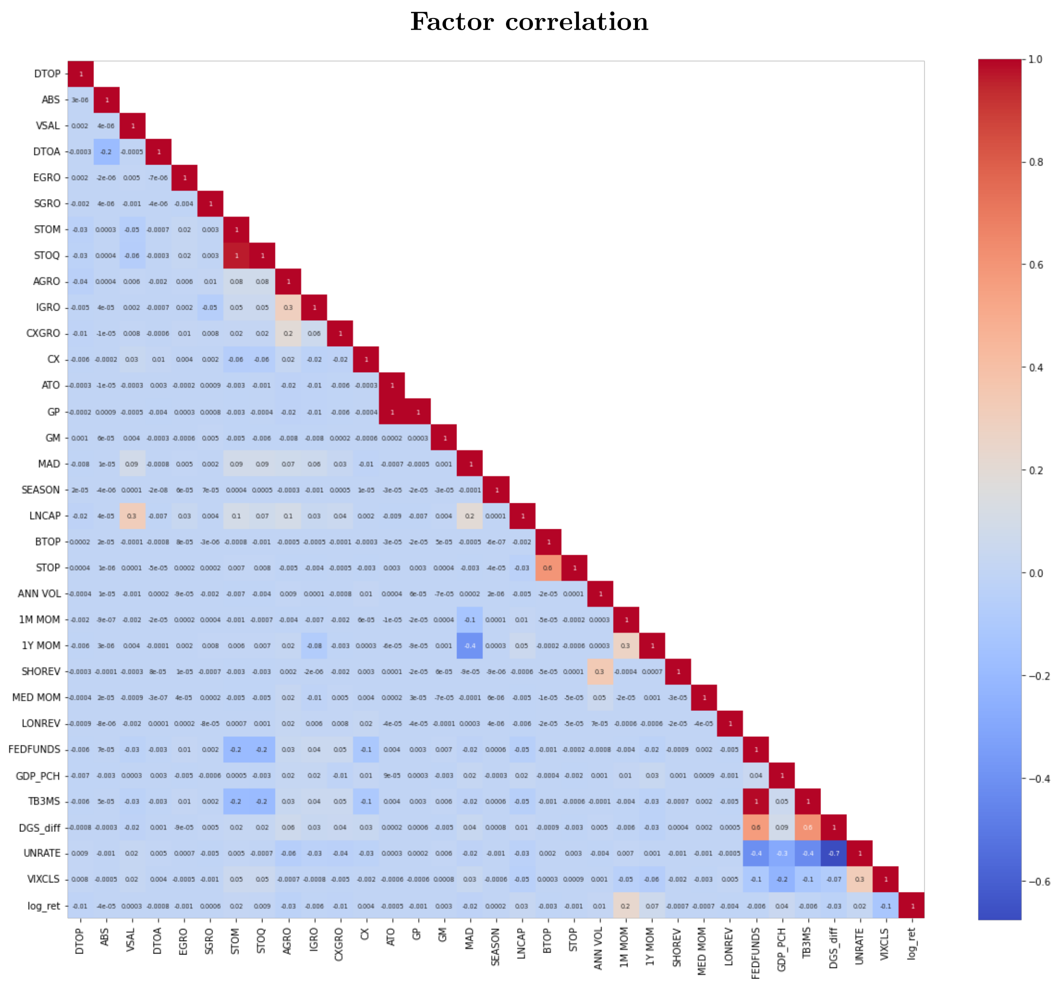


Figure 4.3: *How all factors are correlated. Source: Own work.*

since STOQ is just an "average" of three months of STOM. Thus we are able to remove one of these without impacting the model negatively. We remove STOQ. Next up is GP and ATO. If we take a look at how these are calculated, this is maybe not totally surprising

$$\text{ATO} = Sales/TA,$$
$$\text{GP} = (Sales - COGS)/TA.$$

We see that the only difference is, that we in GP subtract $COGS$ from $Sales$. This is apparently not enough to make a difference, and thus we can remove one. We remove ATO. Lastly, we consider FEDFUNDS and TB3MS. Taking a look back at Figure 4.2, it is clearly seen, that these two macro factors are highly correlated. While they do not share the exact same values, they follow the same trend almost one-to-one. Since they also have a correlation of one, they give the exact same information to the model. Thus, we remove TB3MS. Now with redundancies removed, we have a (hopefully) more effective dataset.

4.4 Issues with data collection

With all these factors and the macroeconomic variables, we are now almost ready with our final dataset. As we mentioned above, our dataset consists of a mixture of quarterly and monthly factors. This will naturally be the cause of some issues when creating a dataset. Our goal with this thesis is to predict monthly log returns, however, many of the fundamentals used are not available on a monthly basis. This left us with a few options. Either we could shift our focus to quarterly log returns, drop the quarterly factors, or work with a combination of quarterly and monthly data. Focusing on quarterly log returns leaves little to the practicality of our results while dropping quarterly factors leaves us with a fraction of our original dataset. Thus we decided to go with the final option. To circumvent the issue, we have copied all quarterly factors so they fill each month in their corresponding quarter, making a pseudo-monthly dataset.

Because of inconsistencies in the data, we ran into many instances of not-a-number (NaN) values caused by missing data and INF or -INF (infinity values as a result of dividing with 0 in numpy) because of random occurrences where the close price for some companies was set to 0. These issues are resolved in the data pre-processing section following. The last issue was companies with such inconsistent data, that they were hardly usable. We ran into such two companies with the tickers: CRGE & HYMC. Because of the size of our dataset, we can remove these two companies.

4.5 Data Pre-Processing

Combining all the information from above we can now create a monthly data set. We include all factors since we want to use them as features in our neural network. Furthermore, we include our macro data and include these as features as well. We get a total of 30 features which we use in our neural network. Before any of the factors were computed, we made some assumptions about the data. When running the neural network, we need to make sure that no entries in the data have a NaN value, INF, or -INF. To ensure this, we first drop all data points which have NaN log returns since it makes no sense to include those. When going through the data, some quarterly data were missing. In these cases we filled the NaN values with the previous data. This was done for assets total, earnings per share, shares outstanding, and sales turnover. Doing this, we did not have any cases where the factors would have an INF value. We sorted the dataset with respect to the tickers' alphabetical order and afterward sorted on dates for each ticker. Finally, we computed the log returns for each month. This log return was then shifted, such that the model predicts one month ahead stock log returns $[t+1]$. An illustration of the dataset can be viewed in Table 4.3. This is a very small sample of our dataset since this only includes 11 datapoints[9]. Looking at the table, we observe the shifted log returns denoted as "Target", which is the log return at time $[t+1]$ After the dataset has been created we now divide our dataset into a training

DATE	TIC	ABS	DTOA	LNCAP	$\cdots$	UNRATE	Log returns	Target
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
2022-08-31	A	-0.007631	0.7750	24.3063	$\cdots$	0.037	-0.053656	0.129461
2022-09-30	A	-0.001804	0.7654	24.4358	$\cdots$	0.036	0.129461	0.113509
2022-10-31	A	-0.006931	0.7654	24.550	$\cdots$	0.035	0.113509	-0.034997
2022-11-30	A	-0.006931	0.7654	24.514	$\cdots$	0.034	-0.034997	0.016107
2022-12-31	A	0.010257	0.7366	24.531	$\cdots$	0.036	0.016107	-0.068791
2016-11-30	AA	-0.010459	0.5568	22.389	$\cdots$	0.047	0.301002	-0.031203
2016-12-31	AA	-0.110507	0.6253	22.359	$\cdots$	0.047	-0.031203	0.260884
2017-01-31	AA	-0.010872	0.6253	22.621	$\cdots$	0.046	0.260884	-0.052377
2017-02-28	AA	-0.010872	0.6253	22.574	$\cdots$	0.044	-0.052377	-0.005508
2017-03-31	AA	0.016983	0.6058	22.570	$\cdots$	0.044	-0.005508	-0.019669
2017-04-30	AA	-0.010483	0.6058	22.550	$\cdots$	0.044	-0.019669	-0.023700
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$

Table 4.3: *Small sample of our dataset. This shows some of our 30 features and our target. The target is the log returns at time* $[t+1]$. *Source: Own work.*

set and a test set. We here choose the constituents of the Nasdaq 100 index for the test set. The Nasdaq 100 index is an index that contains around 100 of the largest domestic and international non-financial companies (NASDAQ, 2023a). All of the stocks are listed on the Nasdaq exchange. They call themself "A leader in performance"(NASDAQ, 2023b, Earnings). Here they compare the index to the S&P500 index (S&P Dow Jones indices, 2023) and conclude that the Nasdaq 100 index has a significantly higher performance. This is one of the main reasons why we want to predict the Nasdaq 100 index. To build the test set, we have downloaded the historical Nasdaq 100 components each year from 2005 to 2022 from the Bloomberg Terminal (Bloomberg L.P, 2023). The test set is then updated each year (1 January) such the true components are in our test set. As described in Figure 4.1, we only include active firms in our dataset, and also only firms that are traded on some of the big exchanges: American Stock Exchange, New York Stock Exchange, and NASDAQ-NMS Stock Market. This influences our test set since some of the firms do not exist anymore and some of the firms are not traded on any of the big exchanges anymore. Thus, we end up with a subset of the Nasdaq 100 index each year. This is shown in Figure 4.4 below. In 2005 the test set only contains approximately half of the components in the Nasdaq 100 index. But as time moves forward we see that the number of components is growing. Thus, our test set

[9] Our full dataset has a total of 1.125.819 datapoints.

grows each year. We make a final modification to our training and test set which is scaling

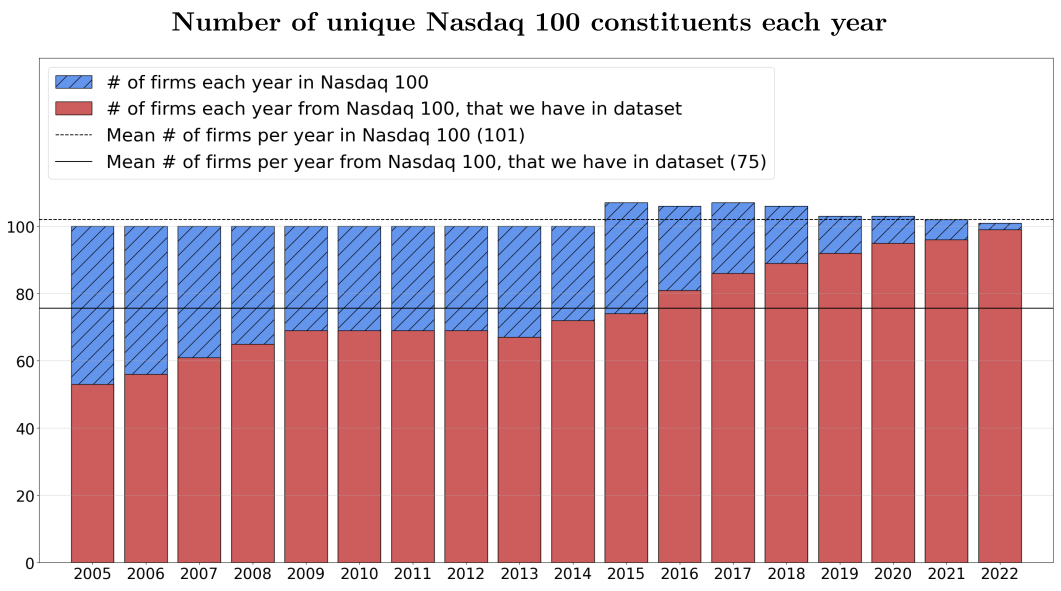


Figure 4.4: *The number of unique firms in the Nasdaq 100 index each year. Red solid bars represent the number of unique Nasdaq 100 firms we have in our dataset each year, while the blue shaded bars represent the total number of unique firms in the Nasdaq 100 index each year. The mean number of firms is also shown (value in legend). Source: Own work.*

the features. Scaling features allow the model to converge much faster (Ioffe and Szegedy, 2015) which is very useful when having a large dataset. We use a MinMax scaler from the Scikit-learn library for Python to scale our features. This is calculated in the following way,

$$X_{std} = \frac{X - min(X)}{max(X) - min(X)}, \qquad \text{fit(data)},$$

$$X_{scaled} = X_{std} * (max - min) + min, \qquad \text{transform(data)}.$$

We only use $fit\_transform()$ on our training set since we do not want the training set to be biased of the test set. Lastly, we also make sure that we make a new scaler for each feature such that the size of different features do not affect the others. The code is shown in Figure 4.5

```python
for col in train.columns[2:-1]: #2: this is tic, datadate. -1: Don't scale target
    scaler = MinMaxScaler()
    train[[col]] = scaler.fit_transform(train[[col]])  #Use fit_transform on training set
    test[[col]] = scaler.transform(test[[col]]) #Use transform on NASDAQ COMPANIES
```

Figure 4.5: *Feature scaling. We here observe how each feature is scaled. We ignore the first two rows since these are not features. We also don't scale the target since this could reveal the log returns we want to predict in the test set. Source Own work (snapshot from python code).*

Input size setup

When predicting stock log returns, there are multiple approaches that need to be taken into consideration. Probably the simplest approach is to use a single data point, to predict a given stock log return. An example of this, we want to predict A's log return on 2022-11-30. For doing this, we only use the feature data for that exact date to predict A's log return. This approach is very simple since it has no memory of A's previous data points, making this a "naive" prediction. Thus, we want to input some sort of memory, which we call **input size**. Our input size has a predefined memory length which is set to 12 months. This means that we use a total of 12 months of data (12 data points) to make a prediction of the stock log return. In this way, we use 12 times more data to predict a single stock log return and hence, the model also trains on 12 times more data. Table 4.4 shows a snapshot from our dataset where the data points for A end and the data points for AA begin. We here illustrate the idea of input size, by illustrating an input size of three months. We predict the current month's log return using this and the two previous months' data points. It is also worth noticing that the input size only refers to the stock's unique ticker e.g. A only has a memory of A itself and not other tickers etc.

Furthermore, looking at Figure 4.4, there are no log returns included when the input size period is too short (shorter than 12 months in our case and shorter than three months in the figure). This can be seen when the ticker changes from A to AA. Since the two first data points for AA do not have two previous data points, we ignore the log return for these data points. Lastly, when there are too few data points for a single firm (less than three in this example), we do not predict any log return on this given stock. We can now have a look at how this was implemented in our code. This is shown using pseudo code. The first algorithm is used for both the training set and the test set, whereas the second is only used for the test set. The main idea of these algorithms is to separate all data such that we can access it month by month. This idea originates from our way of training and testing.[10] Each month contains the current month's data for all stocks, and also two periods back as

[10] See alternating training and testing Section 5.1.

DATE	TIC	ABS	DTOA	LNCAP	···	UNRATE	Target
⋮	⋮	⋮	⋮	⋮	…	⋮	⋮
2022-07-31	A	-0.007631	0.7750	24.3063	···	0.037	0.139942
2022-08-31	A	-0.001804	0.7654	24.4358	···	0.036	0.120202
2022-09-30	A	-0.006931	0.7654	24.550	···	0.035	-0.032940
2022-10-31	A	-0.006931	0.7654	24.514	···	0.034	0.016238
2022-11-30	A	0.010257	0.7366	24.531	···	0.036	-0.066478
2022-12-31	A	-0.006136	0.7366	24.462	···	0.036	0.351213
2016-11-30	AA	-0.010459	0.5568	22.389	···	0.047	-0.030721
2016-12-31	AA	-0.110507	0.6253	22.359	···	0.047	0.298077
2017-01-31	AA	-0.010872	0.6253	22.621	···	0.046	-0.051029
2017-02-28	AA	-0.010872	0.6253	22.574	···	0.044	-0.005493
2017-03-31	AA	0.016983	0.6058	22.570	···	0.044	-0.019477
2017-04-30	AA	-0.010483	0.6058	22.550	···	0.044	-0.023421
⋮	⋮	⋮	⋮	⋮	…	⋮	⋮

Table 4.4: *input size illustration. The table illustrates an input size period of three months, where three data points are concatenated to predict a single stock log return. By doing this, we obtain three times more data and the model gets a higher prediction accuracy. Please note that all data points in the memory are from the same stock and we do not include any log returns which do not have two previous data points. This can be seen when the ticker changes from A to AA. Source: Own work.*

shown in Figure 4.4. For the test set, we use the second algorithm since this also keeps track of the tickers which are in the Nasdaq 100 index that current year. After the algorithm has allocated the rightful tickers to each year, the algorithm then calls the first algorithm for creating the input size in the test set. Using these algorithms, we might end up in a situation where we have an inconsistent amount of data points for some stocks in some months. These we remove using another algorithm defined in Appendix 2[11] as "remove_input_size_errors".

Algorithm 1 Input size date allocation

```
input_size ← x                                   ▷ Define input size of length x
IS_data ← [ ]                                    ▷ Allocate storage for all months
months ← unique_dates(data)                      ▷ Get all dates in dataset
for $i$ ← (input_size − 1) to months lenght do
    IS_data[i] ← data[months[i-input_size+1]:months[i]]
end for
Return IS_data
```

[11] From line 189 until 237.

Algorithm 2 Test set modification and input size

```
IS_data_test ← [ ]                              ▷ Allocate storage for all months
years ← [test years]                            ▷ List of all years in test set
for $i$ ← year[first] to year[last] do
    temp ← data[year,test tickers]              ▷ Only assign tics for that given year
    IS_data_test[i] ← Algorithm_1(temp)
end for
Return IS_data_test
```

5 Model setup

This section explains how every part of the model is set up and implemented. First, an explanation of how we train and test the model since we use an approach that is somehow unique. Second, the hyperparameters are tuned using a big grid search containing a total of 1296 models. Afterward, we sort out parts of the grid that are suboptimal for minimizing the computational cost. This leaves us with a total of 16 models. Third, the model creation with an algorithm combining both optimal hyperparameters and alternating training and testing is shown. Fourth, a walkthrough of 2 different types of portfolios, long/short and buy/hold. Lastly, we show calculations done for portfolio measurements.

5.1 Alternating training and testing

When training and testing a deep neural network, there are multiple approaches one could use. The standard way is to split the dataset into a train and a test (80% / 20%). We use a slightly different approach since we want to predict the stock log returns of the constituents of the Nasdaq 100 index. To do this properly, we use an approach similar to multiple research papers, where alternating training and testing are implemented (Dixon and Polson, 2020; Nakagawa et al., 2018). Figure 5.1 visualize how we have implemented it. We want to train a unique neural network model for each 12 months of data. Therefore we iterate through our entire data sliding one month ahead, giving a total of 384 iterations and thus 384 unique models. Looking at the figure, the training begins in February 1990. We then build a model for each month from January 1990 until December 2004. In this period we only fit the data to the unique models. From January 2005 we start to make predictions on the test set using a unique model which is created for each month. As mentioned we test our model on the Nasdaq 100 index in the last month in every model. The Nasdaq 100 index is indicated as the shaded area in the last month within the testing period. The main idea for doing alternating training and testing originates from the idea that it should be easier to predict the $[t+1]$ log returns for Nasdaq 100 if we train the model on $[t+1]$ log returns for a lot of other stocks. By doing this, we can assume that our neural network captures some of the trends in the present month, and thus make a better prediction on the test set.

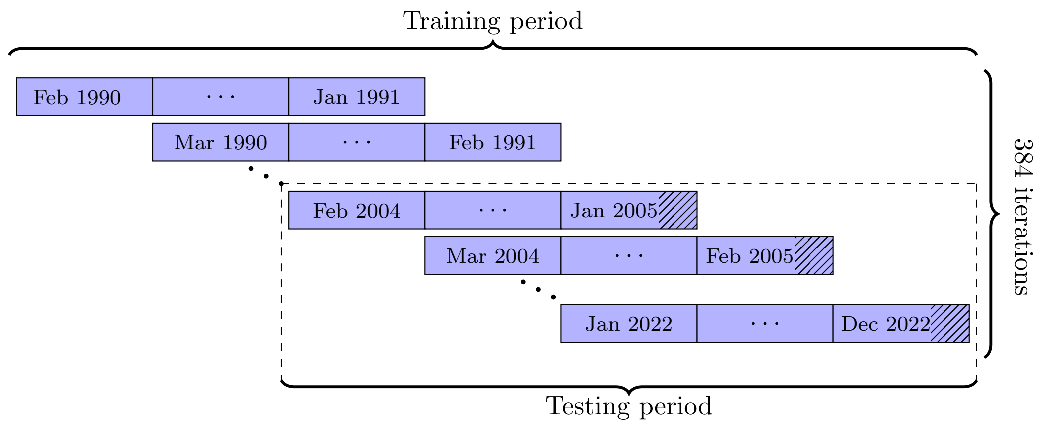


Figure 5.1: *Alternating training and testing. From January 1990 until December 2004 we only fit the data. After 2005 until December 2022 we fit the data but also predict the log returns for the corresponding test set. The shaded area indicates the test set which lies within the last month. Source: Own work*

5.2 Hyperparameters

Hyperparameters are an essential part of training a neural network. These are chosen before we initialize the model and do not change throughout the training phase. The hyperparameters we are choosing are mostly the architectural design of the neural network. The idea is to choose the hyperparameters which minimize the validation error (MSE) the most and use that given model to predict the test set.

Since we do not have a single model in our framework, the task is to find the optimal hyperparameters for each model hence, each month. This is due to the usage of alternating training and testing. Since the optimal hyperparameters are almost impossible to guess, we will be using a grid search for finding the optimal hyperparameters. Grid search is a method in which we keep all hyperparameters fixed except one. By doing this, we can examine how the hyperparameter changes affect our model. This is a very simple way of optimizing hyperparameters which can lead to some challenges. The easiest challenge to think of is a case where the optimal hyperparameters are not defined in the grid. Then the grid search would only find a sub-optimal model instead of an optimal model. This can of course easily happen since we choose our grid manually. This challenge can be countered if the grid is large enough e.g. have a grid of so many points that we can't miss the optimal hyperparameters. This is of course not feasible since this would require an unlimited computational effort. We, therefore, choose a much simpler strategy that requires a lot less computational effort. The strategy is described as:

1. Choose a random month and perform a big grid search to narrow down the hyperparameters. The month should be chosen outside of a recession/ market crash period.

2. From the large grid, sort out unusable hyperparameters and create a much smaller grid with usable hyperparameters.

3. Run the small grid on ALL months.

This strategy should increase the chance of finding optimal hyperparameters for every model. When defining the large grid, we need to be aware of the risk of overfitting. If we tune our hyperparameters too much with respect to the training set, the model will likely overfit which will give a very low training MSE. This could happen if we increased epochs too much, increase hidden layers etc. Keeping this in mind we define the large hyperparameter grid as the following.

- **Neurons** [25, 50, 100]
- **Epochs** [50, 75, 100, 150]
- **Learning rate** [0.001, 0.0005, 0.0001]
- **Dropout** [0.01, 0.05, 0.1]
- **Batch size** [32, 64, 128]
- **Hidden layers** [1, 2, 3, 4]

We run the large grid search in the month of July 2019. We use the same algorithm as we use for training/testing each month. This algorithm is described in the model section. Doing this, we get a total of $3 \times 4 \times 3 \times 3 \times 3 \times 4 = 1296$ models. In the table below, we have shown the top 10 models which minimize the MSE of the validation set the most. The

Rank	Mse Val	Neurons	Epochs	Learning rate	Dropout	Batch size	Hidden Layers
1	0.014739	50	50	0.0005	0.05	64	2
2	0.014760	50	50	0.001	0.05	64	2
3	0.014854	50	50	0.001	0.01	32	1
4	0.014883	50	100	0.0001	0.1	128	1
5	0.015032	50	50	0.0001	0.01	128	1
6	0.015057	100	75	0.0005	0.05	32	1
7	0.015115	100	75	0.0005	0.05	128	1
8	0.015144	100	75	0.0001	0.05	32	1
9	0.015172	50	50	0.001	0.05	32	1
10	0.015182	50	50	0.0005	0.05	128	1

Table 5.1: *Top 10 best-performing hyperparameters in the model for 2019-07-31. The models have been ranked accordingly to the training data's MSE. We chose this month since it was outside of a market crash, it was in our test set, and the returns on the Nasdaq 100 index were relatively flat. This gave a good basis for the large grid, that we could then narrow down for use on all other months. Source: Own work.*

table gives us a clear idea of which parameters are optimal and which are not. By looking at

Table 5.1 we want to define a much smaller grid because of computational cost. Therefore we choose the hyperparameters in the top 3. This gives us a smaller grid defined as:

- **Neurons** [50]
- **Epochs** [50]
- **Learning rate** [0.001, 0.0005]
- **Dropout** [0.01, 0.05]
- **Batch size** [32, 64]
- **Hidden layers** [1, 2]

This yields a total of $1 \times 1 \times 2 \times 2 \times 2 \times 2 = 16$ models per month, which of course has a much lower computational cost than 1296 models. Furthermore, we only optimize our hyperparameters every quarter, this means that we run 16 models for the first month in each quarter, and then use the optimal hyperparameters found for the next two months. This is repeated throughout the whole dataset.

5.3 Model creation

With no further ado, we are now ready to create our model which is an RNN model. This is done using the Python library Keras. Our model is created from the deep learning ideas described in our methodology section. By looking at Figure 5.2, we can examine the python code which contains our RNN model.

First, we define an early stopping that stops our algorithm if we don't have any progress after 50 epochs. Second, we use the checkpoint algorithm which chooses the weights set which minimized the validation loss the most. Both the early stopping and the checkpoints are saved in a callback variable which is used when fitting the model. Third, we create a sequential model, with a single input layer. Fourth, the model uses either one or two hidden layers depending on the input. Firth, the model returns a linear output which is our predicted log returns. All layers use the "ReLU" activation function. Lastly, we use the Adam optimizer and a clipnorm = 1.0 which should prevent clipping challenges. We make it a regression task and use MSE as the measure.

We can now implement alternating training and testing where we each month optimize our hyperparameters using the smaller grid defined in the previous section. Thus, we create Algorithm 4 that does the following,

The algorithm returns a list with a preset number of models for each month. It is then possible to select the model which minimized the MSE the most for each month. In this way, we obtain the most optimal hyperparameters within each month which should increase our prediction accuracy when predicting the test set.

```python
def create_model_rnn(neurons,learning_rate,drop_out,hidden):
    early = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=50) #If no progress in
    ↪  # epochs
    checkpoint = tf.keras.callbacks.ModelCheckpoint("weights.best.hdf5",monitor='val_loss',
    ↪  verbose=0, save_best_only=True, mode='min')
    callbacks_list = [early,checkpoint]
    model = Sequential()
    model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
    ↪  n_features), return_sequences=True))
    model.add(Dropout(rate=drop_out))

    if hidden == 1:
        model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
        ↪  n_features), return_sequences=True))

    elif  hidden == 2:
        model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
        ↪  n_features), return_sequences=True))
        model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
        ↪  n_features), return_sequences=True))

    model.add(SimpleRNN(units = neurons, activation = 'relu', return_sequences=False ))
    model.add(Dense(units = 1)) #Linear output layer

    opt = optimizers.Adam(lr=learning_rate, clipnorm=1.)
    model.compile(optimizer = opt, loss = "mse")
    return model,callbacks_list
```

Figure 5.2: *Keras implementation of RNN in python. In the shown code, we only consider the small grid containing 16 models. To watch the full 1296 models, we refer to Appendix 2, where 3 and 4 hidden layers are implemented as well.*

5.4 Portfolio Preparation

When the alternating training and testing have been done for all months in the dataset, we create two different portfolio strategies. Within these strategies lies a comparison with the Nasdaq 100 index as earlier mentioned. We define three independent periods in which we make comparisons.

- 2005-2011
- 2012-2018
- 2019-2022

The first two periods contain a total of seven years whereas the last period contains a total of four years. By doing this, we can easily observe how our neural network performs in different time periods.

Before creating the two portfolios, we make some assumptions:

Algorithm 4 Algorithm for the monthly grid search. Source: Own work.

```
Optimal ← [ ]                                                  ▷ Allocate storage for grid results
Loss_save ← [ ]                                                      ▷ Allocate storage for loss
for $i$ ← 0 to total months length do
    for $j$ ← 0 to neurons length do
        for $q$ ← 0 to epochs length do
            for $k$ ← 0 to LR length do
                for $p$ ← 0 to Drop length do
                    for $l$ ← 0 to Batch length do
                        for $h$ ← 0 to Hidden length do
                            create.model(neurons[j],epochs[q],LR[k],Drop[p],Batch[l],Hidden[h])
                            model.train(X_train[i],y_train[i])
                            model.predict(X_train[i][80%:])          ▷ Predict validation set to get MSE
                        end for
                    end for
                end for
            end for
        end for
    end for
    Optimal[i] = save(mse,neurons[j],epochs[q],LR[k],Drop[p],Batch[l],Hidden[h])
    Loss_save[i] = model.loss("mse")
end for
Return Optimal, loss_save
```

- **It is possible to buy fractions of stocks**. This makes our framework simple since we don't have to worry about the dollar price of each stock. Even though this is a strict assumption, it should not change the outcome very much.

- **Zero commission**. When buying and selling stocks, we pay zero commission to the stock exchanges. If we were to include commission, our results would vary depending on the number of rebalances we make in our portfolio, but it should not change the main conclusion.

These assumptions should simplify our framework, without changing the outcome. Additionally, we transform the log returns to regular returns for interpretability. Before creating the different portfolios, we make monthly predictions, that we then sort on our predicted returns. A possible outcome could look like Table 5.2.

In the table, we clearly see how all the stocks are sorted on our predicted returns. We then map the tickers to their true returns that we use to calculate the various portfolios' returns, as well as to each ticker's market-cap for the corresponding month, which is used for constructing market-cap-weighted portfolios which we introduce in the following sections.

Ticker	Pred ret	True ret	Market-cap
STX	0.45	0.42	9.27e+09
FLEX	0.27	0.15	7.86e+09
LOGI	0.15	0.17	4.80e+09
AMZN	0.09	0.02	3.06e+10
⋮	⋮	⋮	⋮
ISRG	-0.03	0.01	1.04e+10
BIDU	-0.10	-0.12	7.91e+09
AAPL	-0.24	-0.13	1.47e+11
BB	-0.50	-0.45	6.59e+10

Table 5.2: *An example of how we sort stocks on our predicted returns every month. Source: Own work.*

Long/short portfolio

Fischer and Krauss (2018) created a long/short portfolio in which they go long in the k most undervalued stocks, and short in the k most overvalued stocks. This gives a portfolio size of $2k$ stocks. We choose a similar approach. After we have made the log return prediction at time $[t+1]$, we sort the predictions starting from the most undervalued stocks and going to the most overvalued stocks. Instead of choosing a fixed k, we go long in the top decile and go short in the bottom decile. The strategy is a zero net spend strategy meaning, we are long one dollar and short one dollar.

Within the long/short strategy, we define two different weight strategies, with the first being an equal-weighted strategy. Here we invest an equally large fraction in each stock. Since we have a zero net strategy, we hold a fraction of $w_{i,\mathrm{L}} = w_{i,\mathrm{S}} = 1/k$ in each stock, where L represents a long position and S represents a short position. The second weight strategy is a market-cap-weighted strategy. This is also a zero-net strategy hence, need to make sure that we are long one dollar and short one dollar. We define x_i as the market cap for stock i where we take a long position, and y_i as the market cap for stock i where we take a short position. We can then calculate the fraction we hold of each stock,

$$\begin{aligned} w_{i,\mathrm{L}} &= \frac{x_i}{\sum_i x_i}, \qquad \text{(Long position)}, \\ w_{i,\mathrm{S}} &= \frac{y_i}{\sum_i y_i}, \qquad \text{(Short position)}. \end{aligned}$$

From this, we calculate the monthly portfolio returns as,

$$R_{\mathrm{ls},t} = \sum_i \Big[(w_{i,\mathrm{L},t} \cdot r_{i,\mathrm{L},t}) + (w_{i,\mathrm{S},t} \cdot r_{i,\mathrm{S},t} \cdot (-1)) \Big].$$

This works for both the equal-weighted strategy and the market cap-weighted strategy.

Buy and hold portfolio

The buy-and-hold strategy is a strategy that goes long in all stocks which are undervalued. Thus, we make a prediction of the stock log returns in the test set. We sort out every prediction which is negative hence, sort out every overvalued stock. This allows the portfolio only to invest in stocks that should increase in value. We furthermore design some “rules” for our buy and hold strategy:

- We rebalance the portfolio whenever our portfolio return falls below a certain threshold. Since we are using three different prediction periods, we also use three different thresholds. We use the Nasdaq 100 index average monthly return within our predefined periods. This gives us the following three thresholds $[0.828\%, 1.334\%, 1.480\%]$, which we compare with the portfolio's monthly return. When rebalancing, we make a new prediction and buy all the undervalued stocks.

- Whenever our portfolio performance is above our threshold, we hold the stocks in our portfolio. We set up a condition for our portfolio, that specifies, that even though the current return beats the Nasdaq 100 threshold, we are still careful, and check if next month has any positive predictions. If not, we sell our portfolio and go cash to avoid a sudden crash.

  Using our approach, it could happen that we hold the first-month portfolio throughout the whole period if this portfolio outperforms the average monthly Nasdaq 100 index return every month.

- Whenever we fall below the given threshold and do not have any positive predictions, we sell the portfolio hence, we only hold cash. The reason for this is that we do not want to hold anything if we predict a market crash. The main objective is to hold cash throughout the two big crashes that occur in the span of our test set.[12]

- We define three different rebalancing frequencies: 1 month, 3 months, and 6 months. The idea is to compare the results when we rebalance our portfolio at different timespans. This framework makes it more realistic since not all investors rebalance their portfolios each month.

We furthermore introduce two weight strategies with the first being an equal-weighted strategy. After the model has made a prediction and chosen all, k, undervalued stocks, we invest an equal amount in each stock $1/k$. Our other strategy is a market cap-weighted strategy. We define x_i as the market cap for stock i where we take a long position. Thus,

$$w_i = \frac{x_i}{\sum_i x_i}.$$

[12] The Great Recession and the COVID-19 recession.

From this, we calculate the monthly portfolio returns as,

$$R_{\mathrm{bh},t} = \sum_i \Big[w_{i,t} \cdot r_{i,t} \Big].$$

This works for both the equal-weighted strategy and the market cap-weighted strategy.

Portfolio results

Using both the results from our long/short strategy and the results from our buy and hold strategy, these can now be compared with the Nasdaq 100 index. We compare the total cumulative return with the important note that we do not take compounding into consideration. Thus, this total return is simply the sum over all returns

$$R_{\text{pf total return}} = \sum_t R_{\mathrm{pf},t},$$

where $R_{\mathrm{pf},t}$ denotes the return of the chosen portfolio[13] for month t. Since we do not take compounding into consideration, we can simply divide this total return by the number of years, to find the average annual return.

Afterward, we calculate the historical volatility of our portfolio.

$$a = \frac{1}{T} \sum_t^T \ln(1 + R_{\mathrm{pf},t}),$$

$$\sigma_{\mathrm{pf}} = \sqrt{\frac{\sum_t^T \Big[(\ln(1 + R_{\mathrm{pf},t}) - a)^2 \Big]}{T - 1}}.$$

We then multiply this number with $\sqrt{12}$, to get annual volatility. Lastly, we compute the Sharpe ratio as

$$\mathrm{SR} = \frac{R_{\mathrm{pf}} - r_{\mathrm{f}}}{\sigma_{\mathrm{pf}}}$$

[13] Either long/short or buy-and-hold portfolio.

6 Results

This section will go through all the results that we obtained in this thesis. First, we go through all the portfolio results which compare our buy-and-hold portfolios with the Nasdaq 100 index. This is done in all three predefined periods, 2005-2011, 2012-2018, 2019-2022. In addition, we calculate volatility and Sharpe ratio for all portfolios to compare performance. Furthermore, we investigate the long/short portfolio returns for every year in our test set. Second, we go through the results which are directly related to the model tuning. This includes an analysis of hyperparameters used in good and bad times. In addition, we compare the training and validation loss in good and bad times.

6.1 Portfolio results

After implementing the different trading strategies for our three different periods, we are ready with our results. As stated numerous times before, our goal for this thesis is to predict the monthly log returns of the constituents of the Nasdaq 100 index. We use these predictions as a basis for the management of our portfolios. In total, we have created four different portfolios using two different strategies each with two different weighting rules. These portfolios are:

- Buy-and-hold portfolio with every component equally weighted.
- Buy-and-hold portfolio with every component weighted after their market cap.
- Long/short portfolio with every component equally weighted.
- Long/short portfolio with every component weighted after their market cap.

We compare the cumulative returns of every portfolio with the cumulative returns one could obtain if they just invested in the Nasdaq 100 index. Firstly, we explore how the buy-and-hold portfolios perform with different rebalancing frequencies. That is, we set up three different pairs of portfolios, where we in the first one are able to rebalance our portfolio every month, be it going cash, selling and buying assets, or just holding everything for the next month. In the next pair, we are able to rebalance our portfolio every third month, and will in the other months be forced to hold the assets up until the next rebalancing point. The last is where we are only able to rebalance every sixth month. In the following figures, the market cap-weighted portfolios are represented by dotted lines, while equal-weighted portfolios are represented by solid lines. The rebalancing points will be marked by circles for the equal-weighted portfolios and stars for the market cap-weighted portfolios, while the portfolios that are rebalanced every month do not need these markers. An important note here is that in this thesis we do not consider the compounding effects of investment, and thus we only focus on the raw cumulative monthly returns, giving us a clear basis for comparing different types of portfolios.

First up in Figure 6.1 is the period from 2005 up until 2011 containing the Great Recession in the period from December 2007 to June 2009. Overall in the figure, we notice

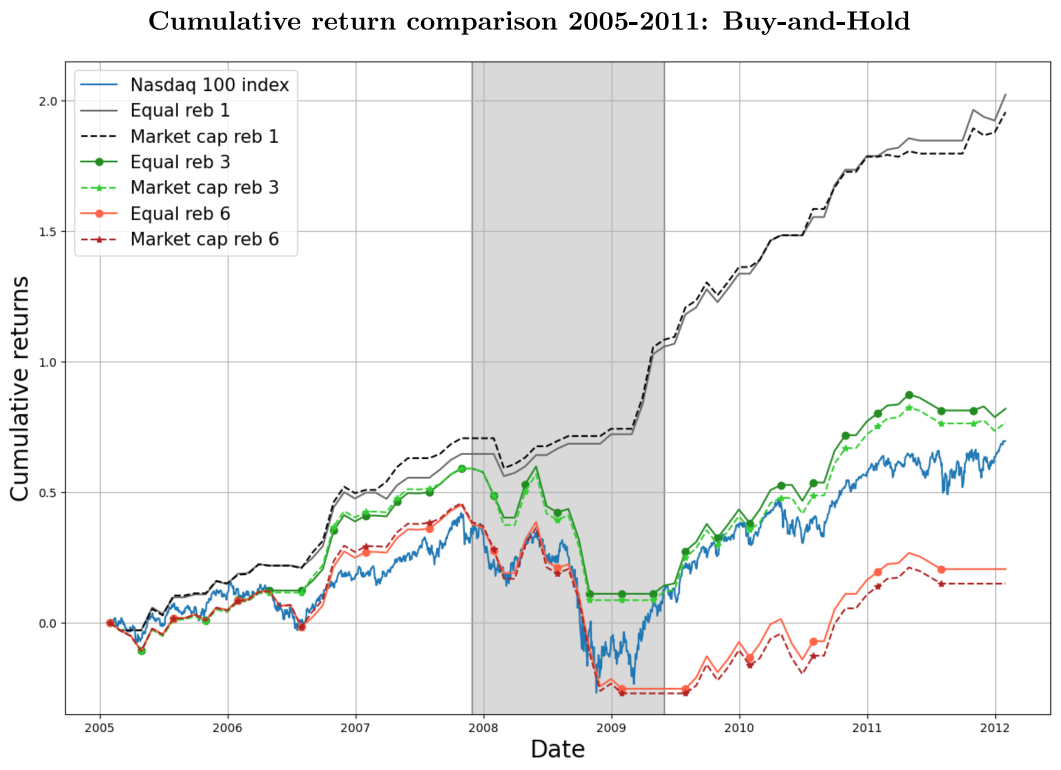


Figure 6.1: *Cumulative return comparison in the period 2005-2011. The plot shows a comparison of the Nasdaq 100 index (blue) with our buy-and-hold strategy with different rebalance frequencies. The shaded area represents the Great Recession: December 2007-June 2009. Source: Own work.*

that the equal-weighted strategies outperform the market cap-weighted strategies slightly. Additionally, we observe that the portfolios that are rebalanced monthly greatly outperform not only the other rebalancing frequencies but most importantly the Nasdaq 100 index. The portfolios that are rebalanced every third month also slightly outperform the Nasdaq 100 index, while the portfolios that are rebalanced every sixth month underperform greatly. Let us explore why that might be. We remember the rules that we set for the buy-and-hold portfolios, and especially the rule where we sell all components and go cash if we predict next month's returns to be negative or if we need to rebalance, but do not have any positive predictions for the current month. In the portfolios that are rebalanced monthly, we are able to (generally) avoid any decreases in cumulative returns, since our predictions tell us when to hold cash. The other portfolios are not so lucky. This is especially prominent during the Great Recession, where we are unlucky, and rebalance right before a big crash. Both rebalancing frequency portfolios suffer because of this. The six-month portfolios are especially bad since they decide that the best thing to do after the crash is to hold cash, and thus miss out on the prominent rise in the market afterward. Overall, rebalancing the portfolios every month yields the best cumulative returns by a long shot. If we were to introduce transaction costs, we might see a tradeoff since it would be more expensive to rebalance every month. Total return in the period and other portfolio parameters are shown in Table 6.1.

The next period is from 2012 up until 2018 shown in Figure 6.2. This period does not contain any recessions and is generally increasing. Again, we notice that the equal-weighted

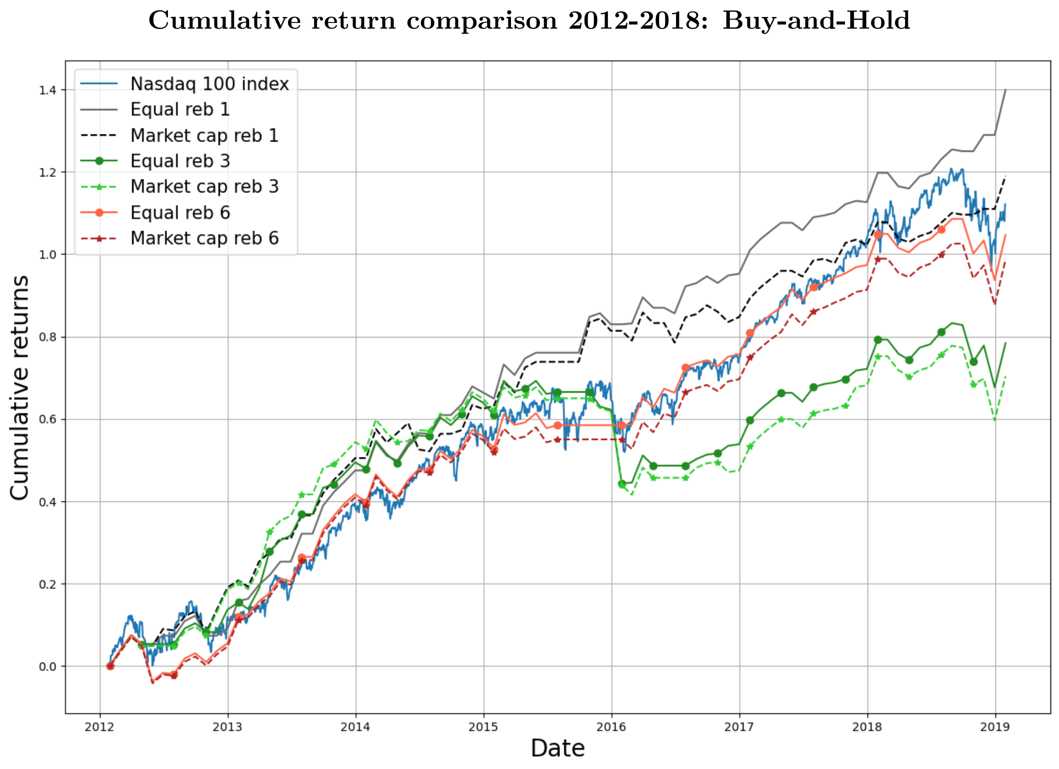


Figure 6.2: *Cumulative return comparison in the period 2012-2018. The plot shows a comparison of the Nasdaq 100 index (blue) with our buy-and-hold strategy with different rebalance frequencies. Source: Own work.*

strategies outperform the market cap-weighted strategies. this time we see a more prominent difference in the two portfolios that are rebalanced every month. This larger difference arises because of two factors. Firstly, it is clear that our model struggled a bit in this period, which is interesting since increasing markets should be generally easier to predict. We know that our model struggled a bit in this period because the monthly rebalanced portfolios have a lot of decreases in places where we should have held cash. This tells us, that we predicted some positive movements, that were in reality negative. The second factor that contributes to the difference in equal and market-cap-weighted strategies, is that we most likely have put more weight on some large-cap firms that ended up giving negative returns.

Obviously, the monthly rebalanced portfolios outperform not only the other rebalancing frequencies but most importantly the Nasdaq 100 index, be it not by the same margins as in the previous period. An interesting result in this period though, is the fact that the six-month rebalanced portfolios outperform the three-month portfolios. This seems to have happened by chance: Around mid-2015 both rebalance frequency portfolios decide to go all cash. Then three months later, the three-month rebalanced portfolios decide to invest in something that immediately crashes while the six-month rebalanced portfolios continue to be all cash.

Overall, rebalancing the portfolios every month yields the best cumulative returns again, however not as prominent this time. If we were to introduce transaction costs, we might see that it would not be profitable to rebalance every month. Total return in the period and other portfolio parameters are shown in Table 6.2.

The next period is from 2019 up until 2022 shown in Figure 6.3. This period contains the COVID-19 recession in the period from February 2020 to April 2020. Again, we notice that the equal-weighted strategies outperform the market cap-weighted strategies, this time with a much larger spread for all rebalance-frequency portfolios. This is most likely due to the same reasons as in the previous period: Bad predictions and putting more weight on firms that do badly.

Again, the monthly rebalanced portfolios outperform not only the other rebalancing frequencies but most importantly the Nasdaq 100 index, this time, a little more comfortably than in the previous period. We notice the same pattern here, where we see a lot of dips in places where we should have held cash, telling us, that our model struggled somewhat in this period as well. Fortunately, our model picked up on the crash during the COVID-19 recession, and thus, all portfolios go cash in this period. The portfolios that are rebalanced monthly immediately pick up after the recession, and take off from there, while the three-month rebalanced portfolios do the same when they are able to.

Overall, rebalancing the portfolios every month yields the best cumulative returns yet again, this time more prominently than in the last period. If we were to introduce transaction costs, we might see that it would not be profitable to rebalance every month, at least for the market-cap-weighted portfolio, since this performs significantly worse than the equal-weighted. Again, total return in the period and other portfolio parameters are shown in Table 6.3.

To sum up our findings regarding the buy-and-hold strategy in these three periods, it is easy to see, that to consistently beat the Nasdaq 100 index, one should rebalance

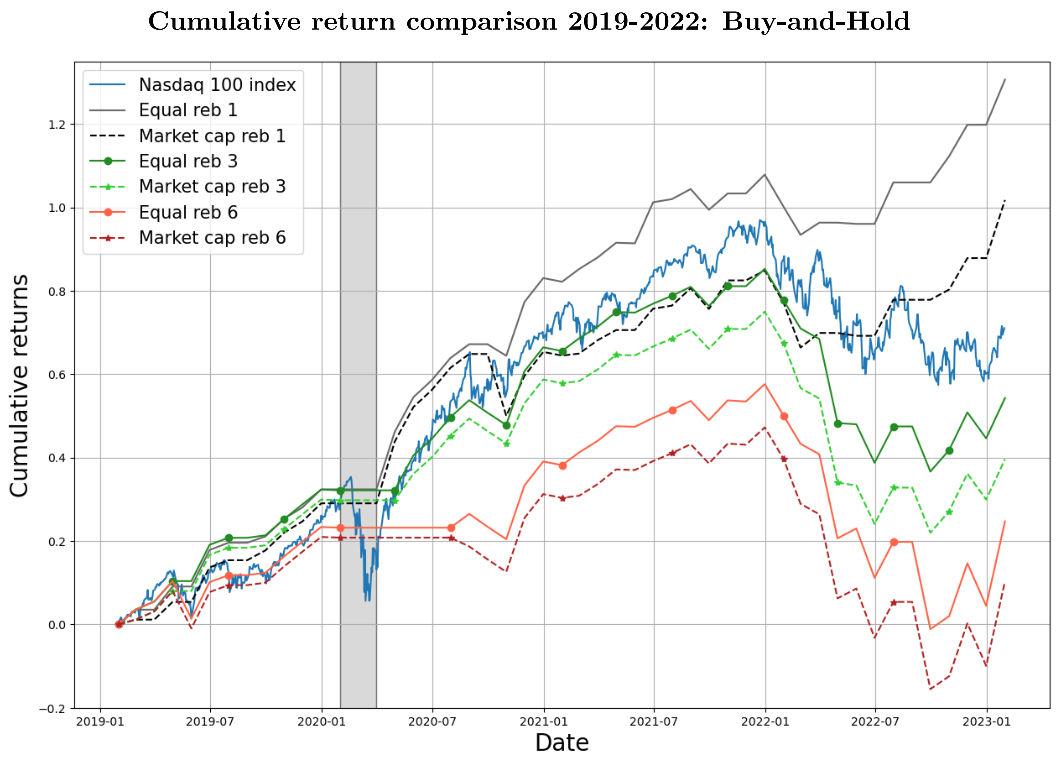


Figure 6.3: *Cumulative return comparison in the period 2019-2022. The plot shows a comparison of the Nasdaq 100 index (blue) with our buy-and-hold strategy with different rebalance frequencies. The shaded area represents the COVID-19 recession: February 2020-April 2020. Source: Own work.*

their portfolio monthly. This result is of course subject to our critical assumption that transactions incur no cost to the investor. For some periods, transaction costs might change our recommendation. Additionally, we observe that equal-weighted portfolios outperform market-cap-weighted portfolios every time. That is, for all different periods, and for all different rebalancing frequencies, it is better to weight all components in the portfolios equally.

The above figures visualize the cumulative return performances of the buy-and-hold strategies within different rebalancing frequencies. Even though this is a good visualization, the figures do not explain the risk within the portfolios. Therefore, we want to take a deeper look at portfolio performance where we compare some different results. We compare total return and an average yearly return, which gives an overall expression of the return performance throughout the period. Even though it seems reasonable to think that, high returns equal a good portfolio, this might not be the case. Therefore we also compare the average yearly volatility. Volatility is often used to measure risk since this indicates the fluctuations of the returns. Hence, high volatility means high fluctuations and vice versa. Even though investors prefer low volatility, this could lead to low returns which is not favored. Thus, we introduce Sharpe ratio. Sharpe ratio measures the relationship between excess return and volatility. Since volatility is always positive, the only way for the Sharpe ratio to become negative is if the excess return is negative. In this case, it does make any sense to invest since you would get a guaranteed return by investing in the risk-free rate. A high Sharpe ratio is generated when excess return is high and volatility is low. Combining these results, we should get a better insight into the profitability of the different portfolios.

Additionally, we include our results from the long/short portfolios. It is worth mentioning that our long/short portfolios are rebalanced every month since we find it very unrealistic not to consider a rebalance every month. This is due to the unlimited liability risk that lies within shorting stocks, where the investor could potentially have a loss that is infinitely large. Therefore we only compare the results with monthly rebalancing (which are also the best-performing portfolios).

We look at the same predefined periods as earlier. The main objective is to observe which portfolios perform better in different periods, not only in the raw returns but also in riskiness. We include the Nasdaq 100 index which we use as a benchmark.

We start by looking at Table 6.1 which shows the profitability from 2005 until 2011. This period contains the Great Recession. Looking at the total return, we clearly see that

Profitability comparison 2005-2011

	Nasdaq 100	Buy-and-Hold eq	Buy-and-Hold mc	Long/Short eq	Long/Short mc
Total return	0.614837	2.022297	1.955062	3.522597	**3.710037**
Avg. Ann. Return	0.087834	0.288900	0.279295	0.503228	**0.530005**
Avg. Ann. Volatility	0.246516	0.150161	**0.149427**	0.276258	0.307028
Sharpe Ratio	0.203138	1.672493	1.616420	**1.684913**	1.603268

Table 6.1: *Profitability of our different portfolios and Nasdaq from 2005 until 2011. All portfolios are rebalanced every month. The numbers which are bold show the best result among all the portfolios. This table shows that the long/short market-cap-weighted has the best return, but also the highest volatility. The lowest volatility can be found in the buy-and-hold market-cap-weighted portfolio. The portfolio with the highest Sharpe ratio is the long/short equal-weighted portfolio closely followed by the buy-and-hold equal-weighted portfolio. Since we are in a period with a financial crisis, we see that the long/short portfolio is performing very well.Source: Own work.*

all portfolios have a higher cumulative return than the Nasdaq 100 index in this period. Additionally, the long/short portfolios strongly outperform both buy-and-hold strategies.

Looking at annual volatility, we observe very high volatilities with Nasdaq and the two long/short portfolios. Comparing this with the buy-and-hold strategies, these are almost half the size. As mentioned, the Great Recession is contained in this period which is why the Nasdaq index has such a high annual volatility. Lastly, we compare the Sharpe ratios. Comparing our constructed portfolios, we observe the equal-weighted portfolios having an almost identical Sharpe ratio and the market-cap-weighted portfolios also having an almost identical Sharpe ratio. Therefore, the two most optimal portfolios are the buy-and-hold equal-weighted portfolio and the long/short equal-weighted portfolio, where it is up to the investor to decide whether more risk compensates for the given return. Finally, the Sharpe ratio for Nasdaq 100 is much lower than any of the other Sharpe ratios, making our constructed portfolios a better choice.

Now looking at Table 6.2 which shows the profitability from 2012 until 2018, we see the highest returns in the long/short portfolios. Comparing the buy-and-hold strategies

Profitability comparison 2012-2018

	Nasdaq 100	Buy-and-Hold eq	Buy-and-Hold mc	Long/Short eq	Long/Short mc
Total return	1.121090	1.398925	1.190133	2.050198	**2.692237**
Avg. Ann. Return	0.160156	0.199846	0.170019	0.292885	**0.384605**
Avg. Ann. Volatility	0.161853	**0.102875**	0.108177	0.198170	0.311077
Sharpe Ratio	0.850238	**1.723497**	1.363291	1.364202	1.163903

Table 6.2: *Profitability of our different portfolios and Nasdaq from 2012 until 2018. All portfolios are rebalanced every month. The numbers which are bold show the best result among all the portfolios. This table shows that the long/short market-cap-weighted portfolio has the best return, but also the highest volatility. The lowest volatility can be found in the buy-and-hold equal-weighted portfolio. The portfolio with the highest Sharpe ratio is the long/short equal-weighted portfolio. This Sharpe ratio is significantly higher than all the other portfolios' Sharpe ratios including the Nasdaq 100. Since we don't have any financial crisis with this period, we see much better results in the buy-and-hold portfolios than the long/short portfolios. Source: Own work.*

with the Nasdaq index, we get total returns that are very close to each other. This is of course also what we observed when looking at Figure 6.2, where the buy-and-hold portfolios only slightly beats the Nasdaq 100 index. By comparing volatility, the two buy-and-hold portfolios have significantly lower volatility than the long/short portfolios and Nasdaq. The highest volatility is observed in the long/short market-cap-weighted portfolio which is much higher than the rest of the portfolios. Lastly comparing the Sharpe ratio, the buy-and-hold equal-weighted portfolio clearly is the most favored, since it has a much higher Sharpe ratio than the rest. Furthermore, all of the portfolios have a significantly higher Sharpe ratio than the Nasdaq 100 index, making our constructed portfolios a better choice.

Now looking at the third Table 6.3 which shows profitability from 2019 until 2022. This

Profitability comparison 2019-2022

	Nasdaq 100	Buy-and-Hold eq	Buy-and-Hold mc	Long/Short eq	Long/Short mc
Total return	0.607360	1.306389	1.017670	**2.069095**	1.195633
Avg. Ann. Return	0.151840	0.326597	0.254417	**0.517274**	0.298908
Avg. Ann. Volatility	0.273535	**0.154577**	0.178277	0.253710	0.408246
Sharpe Ratio	0.487766	**1.993696**	1.323775	1.966239	0.687061

Table 6.3: *Profitability of our different portfolios and Nasdaq from 2019 until 2022. All portfolios are rebalanced every month. The numbers which are bold show the best result among all the portfolios. This table shows that the long/short equal-weighted portfolio has the best return. The two highest volatilities are in the long/short market-cap-weighted portfolio and in the Nasdaq 100 index. The lowest volatility can be found in the buy-and-hold equal-weighted portfolio. The portfolio with the highest Sharpe ratio is the buy-and-hold equal-weighted portfolio. This Sharpe ratio is very close to the long/short equal-weighted portfolio. We observe that the long/short equal-weighted portfolio performs very well during this period which is because of COVID-19 recession. Source: Own work.*

period contains the COVID-19 recession. We see the highest returns in the long/short equal-weighted portfolio. Looking at the table, it is interesting to notice that the second-highest total return is no longer the long/short market-cap-weighted portfolio but the buy-and-hold equal-weighted portfolio. The Nasdaq 100 index again has the lowest return. Comparing annual volatility, we observe the lowest in the buy-and-hold strategies and the highest in the long/short market-cap-weighted portfolio. Furthermore, we observe very high volatility in the Nasdaq index which is because of the large fluctuations in the COVID-19 recession. By comparing the Sharpe ratio, the buy-and-hold equal-weighted and the long/short equal-weighted portfolios have the highest Sharpe ratios. Again, all portfolios outperform the Nasdaq 100 Sharpe ratio.

To sum up all findings in the previous three tables, we clearly see that all of our constructed portfolios outperform the Nasdaq 100 index when comparing the Sharpe ratios. Additionally, when comparing the buy-and-hold portfolios with long/short portfolios we observe mixed results. In the period which does not include any financial crisis, the buy-and-hold portfolios perform significantly better than all other portfolios when comparing the Sharpe ratio. In periods that include financial crises, there isn't much difference between the performance of buy-and-hold portfolios and long/short portfolios.

At this point, we have been through our main portfolio results and shown how the different portfolios perform against the Nasdaq 100 index. We see a clear pattern in our results; the long/short portfolios are clearly the most profitable in terms of cumulative returns compared to the buy-and-hold portfolios and the Nasdaq 100 index. They are however much more volatile. We would like to highlight how the returns in the long/short portfolios are driven. We remember how our long/short portfolios are constructed: We divide all of our predicted returns into ten deciles. We then take a long position in the top decile and a short position in the bottom decile. The cost of the long position is then covered by the yield of taking on the short position, and thus we have a zero-net investment portfolio.

Table 6.4 shows the yearly returns from both the equally-weighted long/short portfolio and the market-cap-weighted long/short portfolio compared to the yearly returns on the Nasdaq 100 index. Additionally, the total return from both portfolios and from the Nasdaq 100 index are shown in each of the three periods.

When looking at the first period from 2005-2011 we notice, that both long/short portfolios are way more profitable than the Nasdaq 100 index. We also notice, that while both long/short portfolios end up giving close to the same return, their yearly returns vary significantly. This is obviously due to how the components of the portfolios are weighted. What is interesting in this period, is how the portfolios behave during the Great Recession. We notice how the Nasdaq 100 index exhibits very negative returns during 2008, while both portfolios yield very high returns during this period. This is of course because of the short positions in both portfolios. If the market is in a crash, a short position will generally yield very large returns. Interestingly, both portfolios yield even larger returns in 2009, which is caused by a combination of the short and long positions.

In the next period, both long/short portfolios are more profitable than the Nasdaq 100 index again, just not to the same degree. In this period, the market is generally increasing, so we do not get the same benefits from the short positions. We do however note, that 2013, 2014, and 2015 yield very large returns in both long/short portfolios. These returns are mainly driven by long positions since the market is generally increasing as seen in Figure 6.2.

In the last period, both long/short portfolios are more profitable than the Nasdaq 100 index again, but this time we also see a significant difference in total returns from the two long/short portfolios. This time, the equal-weighted portfolio is more profitable and yields very positive returns every year. The market-cap-weighted portfolio has very low returns in 2019 and even negative returns in 2020. This negative return happens because of a bad prediction by our model on a stock with a very large market cap, which results in the long/short market-cap-weighted portfolio taking a long position in a stock with very negative returns. This also happens to the equal-weighted portfolio, but not to the same

Long/Short yearly returns 2005-2011			
	Nasdaq 100	Long/Short eq	Long/Short mc
2005	0.087781	-0.074532	0.439022
2006	0.078003	0.342587	0.029099
2007	0.188664	0.349363	0.235497
2008	-0.453377	0.820797	0.627658
2009	0.464035	1.122326	1.235507
2010	0.194748	0.421753	0.706036
2011	0.054982	0.540304	0.437217
Total	**0.614837**	**3.522597**	**3.710037**

Long/Short yearly returns 2012-2018			
	Nasdaq 100	Long/Short eq	Long/Short mc
2012	0.086616	0.097275	0.328551
2013	0.307816	0.503544	0.566637
2014	0.174895	0.498137	0.861772
2015	0.096809	0.540930	0.499000
2016	0.070243	0.126179	0.089809
2017	0.279351	0.104824	0.147985
2018	0.015236	0.179310	0.198484
Total	**1.030968**	**2.050198**	**2.692237**

Long/Short yearly returns 2019-2022			
	Nasdaq 100	Long/Short eq	Long/Short mc
2019	0.245224	0.224020	0.038945
2020	0.456193	0.455907	-0.057037
2021	0.253223	0.491470	0.222668
2022	-0.347280	0.897698	0.991058
Total	**0.607360**	**2.069095**	**1.195633**

Table 6.4: *Yearly returns from long/short portfolios with different weights compared to the Nasdaq 100 index. Total return from the respective period is also shown. Source: Own work.*

degree since the weight for this specific stock is much lower here. In the final year, 2022, the Nasdaq 100 index yields very negative returns, probably as a response to the situation in Ukraine starting in February 2022. Both long/short portfolios respond very strongly to this and yield very large returns for this year.

We have now explored in more detail why the long/short portfolios are so profitable. An important aspect to note, however, is that taking on short positions is a very risky endeavor, and might result in a very large loss. Both portfolios also have a very high volatility as explored above in Tables 6.1, 6.2, and 6.3, which of course is also an indicator as to how risky these types of portfolios are.

6.2 Model Results

During the walkthrough of the portfolio results, we showed how the portfolios performed in all periods. We consider both periods with bad times like the major US recessions and also periods with generally good times. Good and bad times obviously yield varying results in a portfolio sense, since the market will move up and down according to the general economy, but also yield varying results in a prediction sense. This is what we will try to highlight in the following paragraphs.

Firstly, we examine how the hyperparameters are distributed during good and bad times. In Appendix 1 we have shown every third month's model hyperparameters. Remember how we only tune the hyperparameters quarterly, to save on computational power. Then it is just a simple task of counting how many times every hyperparameter appears. We have listed these results in Table 6.5, granted, these are for every month and not every third month, however, the distributions remain the same. We immediately notice, how in good times

Distribution of hyperparameters in good and bad times

Good times

LR	#	%	Dropout	#	%	Batch size	#	%	Hidden	#	%
0.0005	175	50.14	0.01	158	45.27	32	159	45.56	1	173	49.57
0.001	174	49.86	0.05	191	54.73	64	190	54.44	2	176	50.43

Bad times

LR	#	%	Dropout	#	%	Batch size	#	%	Hidden	#	%
0.0005	22	62.86	0.01	16	45.71	32	11	31.42	1	13	37.14
0.001	13	37.14	0.05	19	54.29	64	24	68.57	2	22	62.86

Table 6.5: *We show the distribution of the hyperparameters: Learning rate, dropout, batch size, and the number of hidden layers. These distributions are shown in both good and bad times. Source: Own work.*

the models do not really have preferences in hyperparameters and thus choose roughly the same amount of each hyperparameter. We do observe a small favor towards both the larger dropout parameter and the larger batch size parameter, meaning that these are generally favored by the models in good times.

Jumping to bad times, these are defined as the major US recessions listed in order: "The early 1990s recession" July 1990-March 1991, "The early 2000s recession" March 2001-November 2001, "The Great Recession" December 2007-June 2009, and "The COVID-19 recession" February 2020-April 2020. We immediately notice a change in hyperparameter distribution. The same hyperparameters are still preferred by the models, this time, however, some a much more prevalent. The smaller learning rate parameter is now much more favored. This is also the case for the bigger batch size parameter and the larger amount of hidden layers, while the distribution of the dropout parameter is roughly the same as in good times. These results indicate, that bad times are more difficult to predict and thus require the models to learn more complex patterns given how a smaller learning rate, larger batch size, and more hidden layers are preferred, yielding a more complex model architecture. An important note on these results is that we have a much smaller sample size during bad times of only 35 months (roughly 12 quarters, and thus 12 hyperparameter tunings), meaning that these results should be taken with a grain of salt and should not be found significant. We therefore leave this for further research.

To go along with how hyperparameters change during good and bad times, we also compare the average MSE of the validation set of each month during good and bad times. These results are listed in Table 6.6.

Average validation MSE in good and bad times

	Good times	Bad times
Avg. validation MSE	0.026079768	0.036413190
Avg. validation MSE*	0.026472063	0.032093702

Table 6.6: *Average validation MSE in good times and bad times. The top row is averaged over every month, while the bottom row is only averaged over months where we tune hyperparameters. Source: Own work.*

The top row shows the average MSE over every month of good and bad times. Here, we observe a lower validation MSE during good times, confirming that the models have an easier time during these periods. What is really interesting in this table is the bottom row. Here we have averaged over every third month instead of over every month. The idea behind this is to only look at months where we choose actually optimized hyperparameters. In our current approach of tuning hyperparameters every third month, we might end up choosing suboptimal hyperparameters for the two months in between tunings, which might result in a higher overall MSE than if we tuned hyperparameters every month, which is what we are trying to capture in this table.

Interestingly, for good times, the average MSE remains more or less the same, indicating that these months do not absolutely require optimized hyperparameters to perform well. For the bad times, we observe the opposite. Here we observe a lower validation MSE when averaging over every third month. This means that during bad times, the models perform worse when we do not optimize hyperparameters. We do note the same as with the previous result; we need more data during bad times to find the results significant.

After examining the distribution of hyperparameters in good and bad times, together with the average MSE, we now consider the training and validation loss. As mentioned, we use 20% of our training set as the validation set hence, the idea is to tweak the model according to the validation set, such that the model has a higher prediction accuracy when predicting the test set. Figure 6.4 and 6.5 show how the training and validation loss evolves for every epoch going from 1 to 50 epochs. For every epoch, a mean value is calculated for all models to get the average loss. Loss is measured as the MSE between the true log returns and our predicted log returns. We here examine the loss plot of the optimized models. We start by looking at Figure 6.4, which shows the loss in all the good times i.e. every month not in a recession. Looking at the training loss, we see a clear decrease which means that the model is learning the complex pattern within the training set. This decrease would likely continue if we raised the number of epochs, but then the probability of overfitting would likely increase.[14] Looking at the validation loss, we also see a decrease which means the model is performing better and better when predicting the validation error. Thus, in good times the models are learning the complex patterns within both the training and validation set, hence, they should be able to predict the test set.

Training and Validation Loss in Good Times

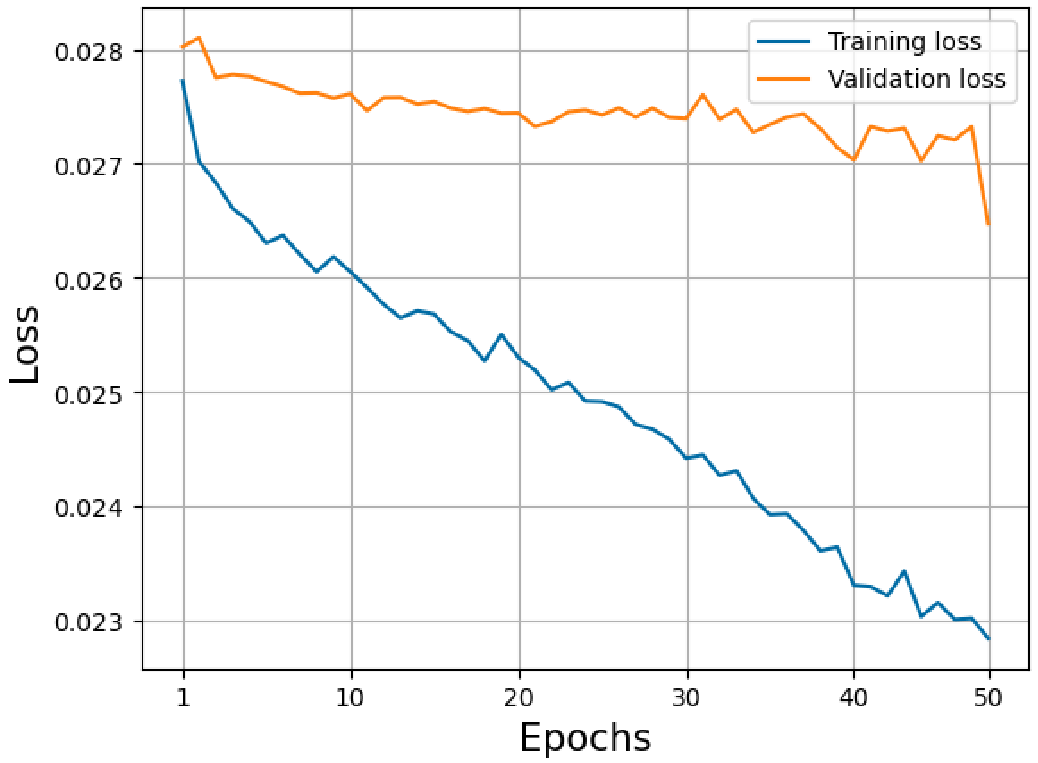


Figure 6.4: *Training and validation loss in all the good times i.e. every month not in a recession. Source: Own work.*

[14] This is shown in the section where we find the optimal 16 models. Here the top 3 models all use 50 epochs, thus this is better when minimizing validation MSE.

Looking at Figure 6.5, we observe training and validation loss in bad times i.e. every month in a recession. The training and validation loss does not behave as smoothly as in Figure 6.4. This is due to the fact that this is the average of 12 models (optimized bad times models) where Figure 6.4 uses a total of 116 models (optimized good times models). Since this figure uses much fewer models, the result is more difficult to interpret. Nonetheless, looking at the training error we see a decrease from the first epoch to the last epoch. This is also true for the validation loss. Thus, in bad times the models seem to be able to learn the complex patterns within the training and validation set. Nonetheless, we do not find these results significant since our dataset only consists of 12 models within bad times. We therefore, as earlier mentioned, leave this for further research.

Training and Validation Loss in Bad Times

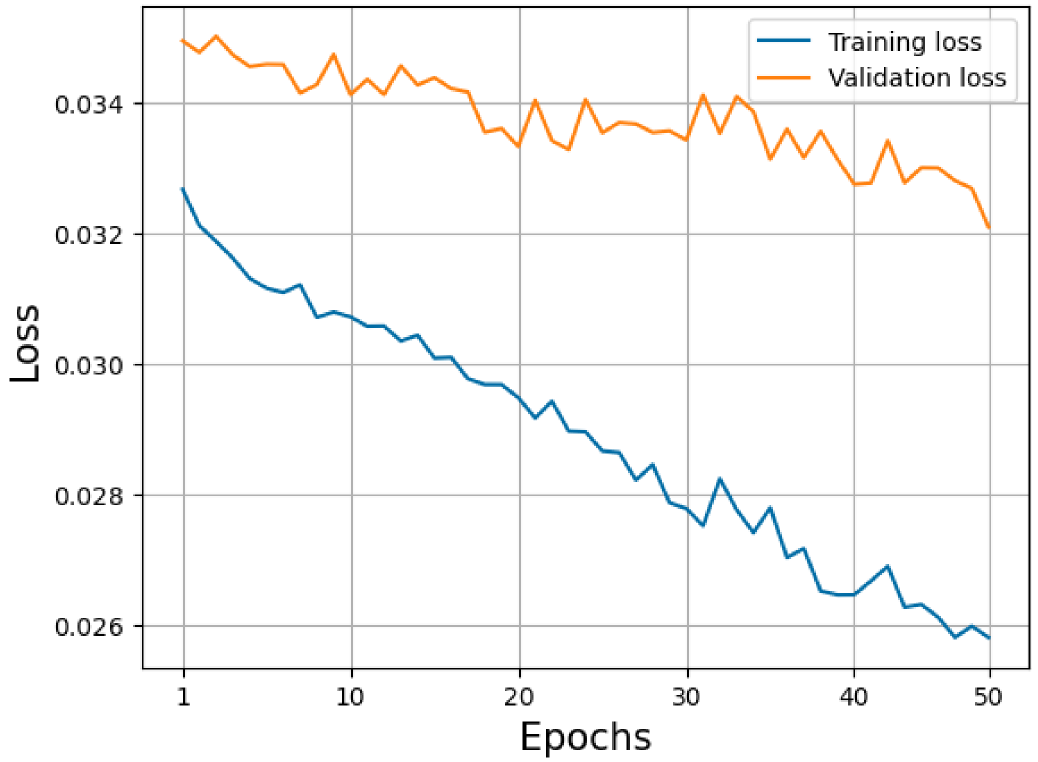


Figure 6.5: *Training and validation loss in bad times i.e. every month in a recession. Source: Own work.*

7 Discussion

Within this section, we discuss different aspects of our approach when predicting the log returns on the Nasdaq 100 index constituents. When creating our neural network, we made several choices that might have affected our final results. We will discuss these and how our neural network is limited by these choices. Furthermore, we try to explain what is expected to happen if we implemented other approaches, etc.

When we prepared the neural network, we also prepared a large grid containing 1296 models which were fitted on a single month, 2019-07-31. Within this single month, we chose the optimal 16 different models to use as a smaller grid search, which found the optimal hyperparameters within every quarter. This approach was mainly done due to limited computational power. If more computational power was available, one could easily optimize the neural network hyperparameters by implementing a large grid search every month. In our results we saw that in bad times, different hyperparameters were needed compared to good times thus, it could be that the needed hyperparameters weren't in our small grid. It is therefore easy to assume that the MSE in bad times would decrease if a large grid was implemented.

Another problem that might arise when using our approach lies within our validation set. Within our training set, we define the last 20% to be our validation set. This is a very simple setting since we do not have any reason to choose the last 20% over the first 20% etc. This issue can be fixed using a k-cross-fold validation method. Here we divide our training data into k subsets. When fitting the model, all k validation samples are used one at a time. Even though this is very easy to implement, it also requires more computational power. For example, using our small grid containing 16 models, every model would need to be run k-times. This would result in a total of $k \times 16$ models. This is also the main reason why this was not implemented. Another approach that requires less computational power is to build a custom validation set. The idea is to build a validation set that is very similar to the Nasdaq 100 index, such that the model is tweaked "in the right direction". This would likely increase the accuracy of the neural network a lot if such a validation set existed and was used. Even though this sound simple, there are many aspects to this. How should they be similar to the Nasdaq 100 index? Should it be on return, market cap, sales, etc.? One approach could be to choose a validation set containing stocks in the S&P 500 index. Even though this sounds fitting, the Nasdaq 100 index writes the following on its webpage: "*the Nasdaq-100 performance surpassed the S&P 500 by a wide margin of index returns between December 31, 2007, and March 31, 2022.*" (NASDAQ, 2023b). This indicates that the S&P 500 could be a bad fit. We, therefore, leave this for further research.

As already briefly discussed in Section 4, we have a lot of quarterly factors in our dataset. This is of course due to the availability of the data needed for the calculations of the factors, that we wanted to include in the dataset. Many company fundamentals were, as previously mentioned, only available on a quarterly frequency, and thus we made a choice to include these when creating the data, and just copy out the resulting factors on the previous two months in the dataset. A possible alternative approach to this issue could have been to be smarter about what factors to include in the sense that we choose factors that have monthly data available. It is unclear how this different approach would affect the results of our models and subsequent portfolios that we tested in this thesis. One could expect the model to be generally more accurate since we would get rid of a lot of quarterly data that might not be truly representative of the company's situation in that specific month. When looking at the portfolio results, it is likely that this change would not have a large effect, since we do not need highly accurate predictions to obtain a profitable trading strategy, as we will discuss shortly.

Staying on the subject of dataset and factors, another decision we made, was to focus on monthly log returns. While this approach is good in a prediction setting, it would be interesting to do the same sort of predictions and portfolios on daily data. We find, that this would yield more applicable results if one was to reproduce these trading strategies in real life. The availability of data of course means that our approach is not possible on a daily basis, however, it remains an interesting point.

Another point is how we might have had advantages, had we used another deep learning model. Throughout this thesis, we have used an RNN to predict stock log returns using sequential data. As we went through in Section 3.3.2, a long-short-term memory (LSTM) model is very similar to that of an RNN, the main difference is that the LSTM model will be able to learn long-term dependencies in the data. It is very likely, that using such a network would yield better prediction results, but it is also highly dependent on how the model uses the data. In this thesis, we only predict one month at a time using the last 12 months as input data. For an LSTM model to be effective, we would likely need to restructure our data, to accommodate a larger input that would help the LSTM model learn the long-term dependencies of the data.

In this thesis, we predict monthly log returns as a regression problem. We then use these predictions as a basis for different trading strategies. Thawornwong and Enke (2004) show that when predicting up and down movements of log returns as a classification problem instead of predicting actual values, one can obtain a better-performing portfolio. Converting our model to a classification model might yield better returns in our portfolios. Combining this with the LSTM architecture, it would be interesting to see what type of results we could obtain, since Fischer and Krauss (2018) show that LSTM outperforms other memory-free classification methods, like DNN, together with other models when predicting directional movements.

For future work, we expect an implementation of an LSTM model on our framework, which could be compared with our results. As mentioned We expect to see improved results due to the memory which lies within the LSTM architecture.

We would also like to have implemented an extension to our portfolio framework. In this thesis, we consider two types of portfolios: Buy-and-hold and long/short. Buy-and-hold performs really well when the market is increasing, while the long/short portfolios are a little more nuanced since we both take on long and short positions. A further implementation could be a hybrid of the two portfolios, where we buy and hold the good-performing stocks and short the bad-performing stocks. With some intelligent rules about managing the portfolio, we believe this could turn out quite profitable.

As another point of further work, one could implement a feature importance analysis. In our thesis, we have used a total of 30 features to predict log returns, but which of these features is most important? We have both implemented company fundamental features which of course are closely related to the log returns of the companies included, but what about our macro features? When looking at Figure 4.3 (Data correlation figure) one can easily observe that the macro data is very uncorrelated with the company factors. Furthermore, we have described how all of our included macroeconomic features can be used when describing the state of the economy. Even though, it could be that our macroeconomic features contribute very little in predicting the correct log return. To address this, we did a feature importance analysis, which is not included in the thesis. The main problem was that since we have so many features, it was very difficult to determine which of the features had the most importance, and which of them had the least. We used the Python library SHAP by Liu and Just (2020), but unfortunately, it could not generate any results. Even if it had provided us with a result, SHAP was very slow when given a lot of sequential data where the limit seemed to be around 50.000 data points which is roughly around 5% of our total dataset. Thus, another library is required for obtaining the importance of each feature.

8 Conclusion

In this thesis, we proposed a deep factor model using the RNN architecture with both company fundamentals and macroeconomic factors. With this model, we managed to predict monthly stock log returns on the constituents of the Nasdaq 100 index from 2005 until 2022. We found that during different economical states, the model accuracy varied, letting us know predictions were easier in good times where we achieved an average MSE of 0.0261, and more difficult in bad times, where we achieved an average MSE of 0.0364.

With these predictions, we were able to use two different trading strategies with different weightings, to construct four different portfolios.

Our framework provides different profitable portfolios using the buy-and-hold strategy. When rebalanced monthly, these portfolios beat the Nasdaq 100 index on cumulative returns in the three pre-defined periods we consider. These are 2005-2011, 2012-2018, and 2019-2022. In all periods, the buy-and-hold portfolios outperform the Nasdaq 100 index on both returns, volatility, and Sharpe ratio as shown in Table 8.1 below. Our framework

Sharpe ratio comparison

	Nasdaq	BH eq	BH mc
2005-2011	0.203138	1.672493	1.616420
2012-2018	0.850238	1.723497	1.363291
2019-2022	0.487766	1.993696	1.323775

Table 8.1: *Sharpe ratio comparison of Nasdaq 100 index, buy-and-hold equal-weighted (BH eq), and market-cap-weighted (BH mc).*. Source: Own work.

also provides different profitable portfolios using the long/short strategy. We find that these portfolios are more profitable when the market is volatile and less profitable in other periods.

When finding optimized hyperparameters we noticed a difference in the neural network's architecture during good and bad times. In good times we see no clear preference in hyperparameters, whereas, in bad times, we see a tilt towards more hidden layers, larger batch size, and lower learning rate. This suggests that bad times are generally more difficult to learn, and thus require a model architecture more capable of learning complex patterns. Nevertheless, we require more data to confirm this result.

References

Abe, M., & Nakayama, H. (2018). Deep learning for forecasting stock returns in the cross-section. *Pacific-Asia conference on knowledge discovery and data mining*, 273–284.

Baldi, P., & Hornik, K. (1989). Neural networks and principal component analysis: Learning from examples without local minima. *Neural networks*, *2*(1), 53–58.

Bayer, J., & Osendorfer, C. (2014). Learning stochastic recurrent networks. *arXiv preprint arXiv:1411.7610*.

Bloomberg L.P. (2023). Nasdaq 100 index, historical components. https://www.bloomberg.com/company

Boyd, J. H., Hu, J., & Jagannathan, R. (2005). The stock market's reaction to unemployment news: Why bad news is usually good for stocks. *The Journal of Finance*, *60*(2), 649–672.

Chen, S.-S. (2009). Predicting the bear stock market: Macroeconomic variables as leading indicators. *Journal of Banking & Finance*, *33*(2), 211–223.

Chollet, F., et al. (2015). Keras.

Cooper, M. J., Gulen, H., & Schill, M. J. (2008). Asset growth and the cross-section of stock returns. *the Journal of Finance*, *63*(4), 1609–1651.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, *2*(4), 303–314.

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., & Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *Advances in neural information processing systems*, *27*.

Dixon, M., & Polson, N. (2020). Deep fundamental factor models. *SIAM Journal on Financial Mathematics*, *11*(3), SC26–SC37.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, *12*(7).

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

Federal reserve economic data - dgs10. (2023). https://fred.stlouisfed.org/series/DGS10

Federal reserve economic data - dgs3. (2023). https://fred.stlouisfed.org/series/DGS3

Federal reserve economic data - fedfunds. (2023). https://fred.stlouisfed.org/series/FEDFUNDS

Federal reserve economic data - gdp. (2023). https://fred.stlouisfed.org/series/GDP

Federal reserve economic data - recession bars. (2023). https://fredhelp.stlouisfed.org/fred/data/understanding-the-data/recession-bars/

Federal reserve economic data - tb3ms. (2023). https://fred.stlouisfed.org/series/TB3MS

Federal reserve economic data - unrate. (2023). https://fred.stlouisfed.org/series/UNRATE

Federal reserve economic data - vix. (2023). https://fred.stlouisfed.org/series/VIXCLS

Fischer, T., & Krauss, C. (2018). Deep learning with long short-term memory networks for financial market predictions. *European journal of operational research*, *270*(2), 654–669.

Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. *Biological cybernetics*, *20*(3-4), 121–136.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

Granger, C. W. J., Morgenstern, O., Granger, C. W., & Morgenstern, O. (1970). *Predictability of stock market prices*. Heath Lexington Books Lexington, MA.

Greenwood, R., & Hanson, S. G. (2012). Share issuance and factor timing. *The Journal of Finance*, *67*(2), 761–798.

Hall, M. (2000). Correlation-based feature selection for machine learning. *Department of Computer Science*, *19*.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., . . . Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, *4*(2), 251–257.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. https://doi.org/10.1109/MCSE.2007.55

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

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

Kelly, B. T., Moskowitz, T. J., & Pruitt, S. (2021). Understanding momentum and reversal. *Journal of financial economics*, *140*(3), 726–743.

Kim, K.-j. (2003). Financial time series forecasting using support vector machines. *Neurocomputing*, *55*(1-2), 307–319.

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

Lachiheb, O., & Gouider, M. S. (2018). A hierarchical deep neural network design for stock returns prediction. *Procedia Computer Science*, *126*, 264–272.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436–444.

Liu, Y., & Just, A. (2020). *Shapforxgboost: Shap plots for 'xgboost'* [R package version 0.1.0]. https://github.com/liuyanguu/SHAPforxgboost/

Maio, P. (2014). Another look at the stock return response to monetary policy actions. *Review of Finance*, *18*(1), 321–371.

Marshall, B. R., Qian, S., & Young, M. (2009). Is technical analysis profitable on us stocks with certain size, liquidity or industry characteristics? *Applied Financial Economics*, *19*(15), 1213–1221.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Jia, Y., Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, ... Xiaoqiang Zheng. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems [Software available from tensorflow.org]. https://www.tensorflow.org/

Medhat, M., & Schmeling, M. (2021). Short-term Momentum. *The Review of Financial Studies*, *35*(3), 1480–1526. https://doi.org/10.1093/rfs/hhab055

Moghaddam, A. H., Moghaddam, M. H., & Esfandyari, M. (2016). Stock market index prediction using artificial neural network. *Journal of Economics, Finance and Administrative Science*, *21*(41), 89–93.

MSCI. (2020). Msci factor analytics brochure. https://www.msci.com/documents/1296102/8473352/MSCI-Factor-Analytics-brochure.pdf/bcb05811-5cad-ab9a-d931-6fa5cbae5e9d

Nakagawa, K., Uchida, T., & Aoshima, T. (2018). Deep factor model. *ECML PKDD 2018 Workshops*, 37–50.

Nasdaq. (2023). https://www.nasdaq.com/

NASDAQ. (2023a). Nasdaq 100 index. https://www.nasdaq.com/solutions/nasdaq-100

NASDAQ. (2023b). Nasdaq 100 index. https://www.nasdaq.com/solutions/nasdaq-100/performance#earnings

Nyse. (2023). https://www.nyse.com/

pandas development team, T. (2020). *Pandas-dev/pandas: Pandas* (Version latest). Zenodo. https://doi.org/10.5281/zenodo.3509134

Park, C.-H., & Irwin, S. H. (2007). What do we know about the profitability of technical analysis? *Journal of Economic surveys*, *21*(4), 786–826.

Pascanu, R., Gulcehre, C., Cho, K., & Bengio, Y. (2013). How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*.

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

Random walk theory. (2012). In *Secrets of the trading pros* (pp. 105–118). John Wiley & Sons, Ltd. https://doi.org/https://doi.org/10.1002/9781119197744.ch7

Resnick, B. G., & Shoesmith, G. L. (2002). Using the yield curve to time the stock market. *Financial Analysts Journal*, *58*(3), 82–90.

Robbins, H., & Monro, S. (1951). A stochastic approximation method. *The annals of mathematical statistics*, 400–407.

Rubbaniy, G., Asmerom, R., Rizvi, S. K. A., & Naqvi, B. (2014). Do fear indices help predict stock returns? *Quantitative Finance*, *14*(5), 831–847.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, *323*(6088), 533–536.

Salem, F. M. (2022). *Recurrent neural networks*. Springer.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, *3*(3), 210–229.

S&P Dow Jones indices, S. (2023). Sp 500 - the gauge of the market economy. https://www.spglobal.com/spdji/en/documents/additional-material/sp-500-brochure.pdf

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

Thawornwong, S., & Enke, D. (2004). The adaptive selection of financial and economic variables for use with artificial neural networks. *Neurocomputing*, *56*, 205–232.

Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop. *COURSERA: Neural networks for machine learning*, *4*(2), 26–31.

Topix. (2023). https://www.jpx.co.jp/english/markets/indices/topix/

Van Rossum, G., & Drake, F. L. (2009). *Python 3 reference manual*. CreateSpace.

Vassalou, M. (2003). News related to future gdp growth as a risk factor in equity returns. *Journal of financial economics*, *68*(1), 47–73.

Wilson, D. R., & Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural networks*, *16*(10), 1429–1451.

WRDS. (n.d.). Wrds. https://wrds-www.wharton.upenn.edu/pages/get-data/compustat-capital-iq-standard-poors/compustat/north-america-daily/

Yegnanarayana, B. (2009). *Artificial neural networks*. PHI Learning Pvt. Ltd.

Zhong, X., & Enke, D. (2019). Predicting the daily return direction of the stock market using hybrid machine learning algorithms. *Financial Innovation*, *5*(1), 1–20.

Appendix

Appendix 1

In the following table, we have listed all optimal hyperparameters within each quarter. Quarters written in cursive represent the major US recessions listed in Figure 4.2.

Date	Val. MSE	Neurons	Epochs	LR	Dropout	Batch size	Hidden
31-01-1991	*0.016952*	*50*	*50*	*0.0005*	*0.01*	*64*	*2*
30-04-1991	0.011282	50	50	0.0010	0.05	32	2
31-07-1991	0.013787	50	50	0.0005	0.01	64	2
31-10-1991	0.009723	50	50	0.0010	0.05	64	1
31-01-1992	0.013806	50	50	0.0005	0.05	32	2
30-04-1992	0.021039	50	50	0.0005	0.05	32	1
31-07-1992	0.012929	50	50	0.0010	0.01	64	1
31-10-1992	0.009308	50	50	0.0010	0.05	64	2
31-01-1993	0.014299	50	50	0.0010	0.01	64	2
30-04-1993	0.018240	50	50	0.0005	0.05	64	1
31-07-1993	0.016122	50	50	0.0010	0.01	64	1
31-10-1993	0.014806	50	50	0.0010	0.05	32	2
31-01-1994	0.008880	50	50	0.0010	0.05	32	2
30-04-1994	0.020600	50	50	0.0005	0.01	64	1
31-07-1994	0.011482	50	50	0.0010	0.01	64	2
31-10-1994	0.008795	50	50	0.0005	0.05	64	1
31-01-1995	0.010009	50	50	0.0010	0.01	64	1
30-04-1995	0.010718	50	50	0.0010	0.05	64	2
31-07-1995	0.014675	50	50	0.0010	0.01	32	1
31-10-1995	0.022397	50	50	0.0010	0.05	64	2
31-01-1996	0.010076	50	50	0.0005	0.05	64	1
30-04-1996	0.020203	50	50	0.0005	0.05	64	2
31-07-1996	0.079758	50	50	0.0005	0.01	32	1
31-10-1996	0.013923	50	50	0.0005	0.01	64	2
31-01-1997	0.029628	50	50	0.0005	0.01	32	2
30-04-1997	0.022207	50	50	0.0010	0.01	32	2
31-07-1997	0.016871	50	50	0.0005	0.01	64	1
31-10-1997	0.015644	50	50	0.0005	0.05	64	1
31-01-1998	0.015910	50	50	0.0010	0.01	32	2
30-04-1998	0.023530	50	50	0.0010	0.05	64	2
31-07-1998	0.038012	50	50	0.0010	0.05	64	2
31-10-1998	0.030064	50	50	0.0010	0.05	64	1
31-01-1999	0.021145	50	50	0.0005	0.01	32	1
30-04-1999	0.015809	50	50	0.0005	0.01	64	1
31-07-1999	0.017587	50	50	0.0010	0.05	64	2
31-10-1999	0.033272	50	50	0.0010	0.05	64	2

Date	Val. MSE	Neurons	Epochs	LR	Dropout	Batch size	Hidden
31-01-2000	0.057385	50	50	0.0005	0.05	64	2
30-04-2000	0.029253	50	50	0.0010	0.05	32	1
31-07-2000	0.022430	50	50	0.0005	0.01	64	2
31-10-2000	0.022533	50	50	0.0005	0.01	64	1
31-01-2001	0.028952	50	50	0.0005	0.05	32	1
30-04-2001	*0.024252*	*50*	*50*	*0.0010*	*0.05*	*64*	*1*
31-07-2001	*0.016961*	*50*	*50*	*0.0005*	*0.05*	*32*	*1*
31-10-2001	*0.020496*	*50*	*50*	*0.0005*	*0.05*	*32*	*2*
31-01-2002	0.016746	50	50	0.0005	0.05	32	2
30-04-2002	0.031684	50	50	0.0010	0.05	32	1
31-07-2002	0.022053	50	50	0.0005	0.05	64	2
31-10-2002	0.034559	50	50	0.0005	0.05	32	1
31-01-2003	0.012735	50	50	0.0005	0.01	64	1
30-04-2003	0.020336	50	50	0.0005	0.01	32	2
31-07-2003	0.014929	50	50	0.0010	0.05	32	1
31-10-2003	0.020493	50	50	0.0005	0.01	32	2
31-01-2004	0.014435	50	50	0.0010	0.01	32	1
30-04-2004	0.016165	50	50	0.0010	0.01	32	2
31-07-2004	0.015388	50	50	0.0010	0.05	32	2
31-10-2004	0.029492	50	50	0.0005	0.01	32	2
31-01-2005	0.012456	50	50	0.0010	0.01	32	1
30-04-2005	0.019225	50	50	0.0010	0.01	64	2
31-07-2005	0.010565	50	50	0.0010	0.05	32	1
31-10-2005	0.026297	50	50	0.0005	0.05	64	2
31-01-2006	0.015522	50	50	0.0010	0.01	32	1
30-04-2006	0.017347	50	50	0.0005	0.05	64	2
31-07-2006	0.013134	50	50	0.0010	0.05	32	1
31-10-2006	0.012203	50	50	0.0010	0.05	32	2
31-01-2007	0.019232	50	50	0.0005	0.05	64	1
30-04-2007	0.015913	50	50	0.0010	0.05	32	2
31-07-2007	0.015730	50	50	0.0005	0.01	64	1
31-10-2007	0.027842	50	50	0.0010	0.05	64	1
31-01-2008	*0.024548*	*50*	*50*	*0.0005*	*0.01*	*64*	*1*
30-04-2008	*0.024769*	*50*	*50*	*0.0005*	*0.05*	*64*	*2*
31-07-2008	*0.022488*	*50*	*50*	*0.0010*	*0.01*	*32*	*2*
31-10-2008	*0.062422*	*50*	*50*	*0.0010*	*0.05*	*64*	*2*
31-01-2009	*0.038353*	*50*	*50*	*0.0005*	*0.01*	*64*	*2*
30-04-2009	*0.061460*	*50*	*50*	*0.0005*	*0.01*	*64*	*2*
31-07-2009	*0.029976*	*50*	*50*	*0.0005*	*0.01*	*64*	*1*
31-10-2009	0.038687	50	50	0.0010	0.05	32	1
31-01-2010	0.021850	50	50	0.0005	0.05	64	2
30-04-2010	0.011278	50	50	0.0010	0.05	64	1
31-07-2010	0.033307	50	50	0.0010	0.05	64	1
31-10-2010	0.020939	50	50	0.0010	0.05	64	2
31-01-2011	0.021359	50	50	0.0005	0.05	32	1
30-04-2011	0.074288	50	50	0.0005	0.05	64	2

Date	Val. MSE	Neurons	Epochs	LR	Dropout	Batch size	Hidden
31-07-2011	0.017796	50	50	0.0010	0.01	32	1
31-10-2011	0.021179	50	50	0.0005	0.05	64	2
31-01-2012	0.023765	50	50	0.0005	0.05	64	1
30-04-2012	0.023941	50	50	0.0005	0.01	64	1
31-07-2012	0.020829	50	50	0.0005	0.01	64	1
31-10-2012	0.014020	50	50	0.0010	0.01	32	1
31-01-2013	0.038859	50	50	0.0010	0.01	32	1
30-04-2013	0.039595	50	50	0.0005	0.05	64	1
31-07-2013	0.017621	50	50	0.0010	0.05	64	2
31-10-2013	0.034894	50	50	0.0010	0.01	64	2
31-01-2014	0.008116	50	50	0.0005	0.01	32	2
30-04-2014	0.018547	50	50	0.0010	0.01	32	2
31-07-2014	0.035413	50	50	0.0010	0.01	64	1
31-10-2014	0.020163	50	50	0.0010	0.01	64	2
31-01-2015	0.049099	50	50	0.0005	0.01	64	1
30-04-2015	0.018581	50	50	0.0005	0.05	64	1
31-07-2015	0.017617	50	50	0.0005	0.05	32	1
31-10-2015	0.024561	50	50	0.0010	0.05	32	1
31-01-2016	0.027386	50	50	0.0010	0.05	32	2
30-04-2016	0.016783	50	50	0.0010	0.05	32	2
31-07-2016	0.021735	50	50	0.0010	0.01	64	2
31-10-2016	0.023111	50	50	0.0010	0.01	32	1
31-01-2017	0.017895	50	50	0.0005	0.01	64	2
30-04-2017	0.022021	50	50	0.0010	0.05	32	2
31-07-2017	0.018824	50	50	0.0005	0.01	32	1
31-10-2017	0.024709	50	50	0.0010	0.05	64	2
31-01-2018	0.029759	50	50	0.0005	0.05	64	1
30-04-2018	0.023069	50	50	0.0005	0.05	64	2
31-07-2018	0.025979	50	50	0.0005	0.01	64	1
31-10-2018	0.030340	50	50	0.0005	0.01	32	2
31-01-2019	0.036451	50	50	0.0005	0.01	64	1
30-04-2019	0.058346	50	50	0.0005	0.05	32	2
31-07-2019	0.041844	50	50	0.0005	0.01	32	2
31-10-2019	0.048993	50	50	0.0005	0.01	32	2
31-01-2020	0.043000	50	50	0.0010	0.05	32	2
30-04-2020	*0.042447*	*50*	*50*	*0.0010*	*0.05*	*64*	*1*
31-07-2020	0.044896	50	50	0.0010	0.01	64	1
31-10-2020	0.053744	50	50	0.0010	0.01	32	2
31-01-2021	0.034729	50	50	0.0005	0.05	64	2
30-04-2021	0.035607	50	50	0.0005	0.05	32	1
31-07-2021	0.101767	50	50	0.0005	0.01	64	1
31-10-2021	0.031751	50	50	0.0010	0.05	32	1
31-01-2022	0.093203	50	50	0.0010	0.01	64	2
30-04-2022	0.068104	50	50	0.0005	0.05	32	2
31-07-2022	0.039314	50	50	0.0005	0.05	32	2
31-10-2022	0.109516	50	50	0.0005	0.05	32	1

Appendix 2

Here we include the main code used in the project.

Import all used packages

```python
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras import layers
from tensorflow.keras.layers import SimpleRNN
from tensorflow.keras.layers import Dropout
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from tensorflow .keras.optimizers import Adam
import keras
from sklearn.metrics import accuracy_score
from keras import optimizers
import matplotlib.pyplot as plt
from tqdm import tqdm_notebook
import datetime
```

Load in NASDAQ 100 data

Load in Nasdaq 100 data that is used in the portfolio section

```python
# Here we calculate the average monthly return for each period we are considering. This is
↪  used for the rebalancing threshold.
Nasdaq100_index = pd.read_csv("NASDAQ100_index.csv")
Nasdaq_avg_ret = []
for i in range(3):
    if i == 0:
        plot_nas = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2005-01-31") &
        ↪  (Nasdaq100_index["DATE"]<"2012-02-01")]["NASDAQ100"].tolist()
    elif i == 1:
        plot_nas = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2012-01-31") &
        ↪  (Nasdaq100_index["DATE"]<"2019-02-01")]["NASDAQ100"].tolist()
    elif i == 2:
        plot_nas = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2019-01-31") &
        ↪  (Nasdaq100_index["DATE"]<"2023-02-01")]["NASDAQ100"].tolist()
    # The data has "." in a lot of places, that we remove
    for x in plot_nas[:]:
        if x == '.':
            plot_nas.remove(x)
    # The dataformat is also strings, so we convert to float
    plot_nas = [float(x) for x in plot_nas]
    # Calculate returns
    ret_nas = []
    for i in range(1,len(plot_nas)):
        ret_nas.append(plot_nas[i]/plot_nas[i-1]-1)
    Nasdaq_avg_ret.append(sum(ret_nas)/np.ceil(len(ret_nas)/21)) # Calculate average return
    ↪  in period
```

Load in and manipulate data

```python
df = pd.read_csv("sorted_dates.csv") # Load in unfinished dataset

df_vix = pd.read_csv("VIXCLS.csv") # Load in VIX (we decided to add it late)
# Make manipulations to VIX df and merge it to dataset
df_vix["datadate"] = df_vix["DATE"]
df_vix["datadate"] = pd.to_datetime(df_vix["datadate"])
df["datadate"] = pd.to_datetime(df["datadate"])
df_vix = df_vix.drop(["DATE"],axis = 1)
df = pd.merge_asof(df,df_vix, on ="datadate")

df_close = pd.read_csv("Closing_price_month.csv") # Load in closing price monthly (we
↪  noticed we might need it)
df_close["datadate"] = pd.to_datetime(df_close["datadate"])
df_close = df_close[["tic","datadate","prccm"]]

df = df.reset_index(drop=True)

df = df.fillna(0) #Fill NaN with 0
# Remove a bunch of unused columns from dataset
df = df.loc[:, ~df.columns.isin(["trt1m","return quarterly","tic.1","ggroup.1", 'ggroup',
↪  "gsector","gsector.1","naics.1","naics","sic.1","sic"])]
df = df.sort_values(["tic","datadate"]) # Sort dataset on tickers and datadate, to get in
↪  correct order
df = pd.merge(df, df_close, on=['tic', 'datadate']) # Merge closing price to dataset
# Remove two firms with wildly inconsistent price and return data
df = df[df["tic"]!="CRGE"]
df = df[df["tic"]!="HYMC"]

df["log_ret"] = np.log(df["prccm"]/df["prccm"].shift(1)) # Include log-return from "current"
↪  month as feature
df["target"] = np.log(df["prccm"].shift(-1)/df["prccm"]) # We want to "forecast" (predict
↪  next month's log-return)

# Load in monthly dataset since we need some information from here for the portfolio stuff
monthly = pd.read_csv("MONTHLY ALL RAW.csv")
monthly = monthly[["tic","datadate","prccm","cshom"]] # Only interested in ticker, datadate,
↪  closing price, and shares outstanding
monthly["mkt cap"] = monthly["prccm"] * monthly["cshom"] # Calculate market cap (used in
↪  market cap weighted portfolios)
monthly = monthly.drop(["prccm","cshom"],axis=1) # After market cap is calculated, we have
↪  no need for closing price and shares outstanding
# Make manipulations and merge to dataset
monthly["datadate"] = pd.to_datetime(monthly["datadate"])
df = pd.merge(df, monthly, on=['tic', 'datadate'])
```

```python
df["mkt cap"] = df["mkt cap"].fillna(0) # We noticed a few places with NaN values in market
↪  cap, remove these


# The next few lines correct an issue we had, where, when calculating log-returns, some
↪  firms' final datapoints would use
# the incorrect data from another firm. This is obviously not correct, so we remove every
↪  firm's final datapoint
df = df.reset_index(drop = True)
temp = [0] # Create temp for storing index
for i in tqdm_notebook(range(len(df)-1)):
    if df["tic"].iloc[i] != df["tic"].iloc[i+1]: # Find where we need to delete a datapoint
    ↪  (where NaN should have appead)
        temp.append(i) # Append the index
        temp.append(i+1)
temp.append(len(df)-1)
df = df.drop(temp) # Remove the index from df
df = df.dropna() # Remove NaN values
df = df.reset_index(drop = True)
df = df.sort_values(["tic","datadate"]) # Sort dataset on tickers and datadate, to get in
↪  correct order, again

n_features = 30 # Choose amount of features for use in later function. We have 31

input_size = 12 # Determine the amount of data for each firm that should på put in the model.
↪  We choose 12 months

df = df[df["datadate"]<"2023"] # We decided for simplicity to cut off the data at 2023

# After examining descriptor correlation, these were deemed not important, and thus dropped
df = df.drop(["MIDREV","MIDREV excess","STOQ","ATO","TB3MS","rf"],axis = 1)
```

Since we decided that we wanted to compare our portfolio results to the Nasdaq 100 index, it makes sense that we build our portfolio on the very same Nasdaq 100 constituents. Thus, we create a test set only containing the firms available in our dataset that were/are Nasdaq constituents in the correct time periods.

```python
#create test train set
from functools import reduce
N100 = pd.ExcelFile("Ticker NASDAQ100 data BLOOMBERG.xlsx") # Load in Nasdaq 100
↪  constituents for every year between 2005 and 2023
Nasdaq100 = {} # Create dictionary that will contain the constituents for each year
# Divide out information from each excel sheet in dictionary
years = np.arange(2004,2024)
Tickers = []
for year in years:
    Nasdaq100[year] = pd.read_excel(N100, str(year))
    for j in range(len(Nasdaq100[year])):
        Nasdaq100[year].at[j,"Ticker"] = Nasdaq100[year]["Ticker"].tolist()[j].split()[0]
    Tickers = reduce(np.union1d, (Tickers,Nasdaq100[year]["Ticker"])) # End up with complete
    ↪  list of Nasdaq 100 constituents


unique_tics = np.unique(df["tic"]) # Determine the unique tickers in dataset
intersect = np.intersect1d(unique_tics,Tickers) # Find the intersection between tickers in
↪  dataset and Nasdaq 100 constituents
train = df[~df["tic"].isin(intersect)] # Create train set with every ticker NOT in Nasdaq
↪  100
test = df[df["tic"].isin(intersect)] # Create test set with every ticker in Nasdaq 100
test_pf = df[df["tic"].isin(intersect)] # Create duplicate test set with every ticker in
↪  Nasdaq 100 for portfolio stuff
train = train.drop(["prccm","mkt cap"],axis = 1) # Remove close price and market cap, since
↪  these are not used for training
test = test.drop(["prccm","mkt cap"],axis = 1) # Remove close price and market cap, since
↪  these are not used for testing
test_pf = test_pf[["datadate","tic","prccm","mkt cap"]] # Only keep datadate, ticker, close
↪  price, and market cap in duplicate test set used for portfolio stuff
```

The next functions are what use the input_size parameter from earlier. Basically, we want to represent the data in ”chunks” of *input_size* (this case 12) datapoints for each firm at a time. Thus, the first function creates a list, *dates_df*, of dataframes for the train set each containing a year worth of data with the next index of the list sliding one month ahead. The final output is as said a list where the index is the corresponding month’s index (dates_df[0] contains the first month and so on)

The bottom function does roughly the same but for the test set. A key difference is, that the resulting list is a level deeper since we need to keep track of specific tickers for each year. Thus the first index corresponds to the year and the next index is the month (dates_df_test[0][0] will be the first month of the first year, while dates_df_test[1][11] is the last month of the next year, and so on)

```python
def input_size_func(df):
    dates_df = [] # Allocate storage to save data for each month
    unique_dates = np.unique(df["datadate"]) # Determine the unique dates
    for i in range(input_size-1,len(unique_dates)): # Iterate over every month but start at
    ↪  "input_size"-1 since we cant look 12 months back starting at month 0
        try:
            # Look at data in range of "input_size" sliding one month ahead at a time
            dates_df.append(df[(df["datadate"]>=unique_dates[i-input_size+1]) &
            ↪  (df["datadate"]<=unique_dates[i])].reset_index(drop=True))
        except:
            None
    return dates_df

def input_size_func_test(dictionary,test_set):
    input_size_test = [] # Allocate storage to save data for each month
    years = np.arange(2019,2023) #'''REMEMBER TO CHANGE!!!''' # Decide which years to
    ↪  iterate over (three respective periods)
    for year in years:
        temp=test_set[test_set["tic"].isin(np.unique(dictionary[year]["Ticker"]))] # Look at
        ↪  specific tickers every year
        temp_test=temp[(temp["datadate"]>=str(year-1)+"-02-01") &
        ↪  (temp["datadate"]<str(year+1) )] # Look at data in range of "input_size" sliding
        ↪  one month ahead at a time
        input_size_test.append(input_size_func(temp_test)) # Use above function for each
        ↪  year, creating a list that will be 1 level deeper
    return input_size_test

dates_df_train = input_size_func(train)
dates_df_test_dict = input_size_func_test(Nasdaq100,test)
dates_df_test_pf = input_size_func_test(Nasdaq100,test_pf)
```

We noticed a little too late on, that the way we created the *input_size* data in the above functions, we did not make sure that we only had data in chunks of 12. Thus, some places would contain a chunk of some random number of data points. This created a bunch of issues, since we need chunks of 12. A quick fix was to create two functions that could remove the inconsistensies.)

```python
def remove_input_size_errors_train(dates_df_train):
    for l in tqdm_notebook(range(len(dates_df_train))): # Iterate over every index of list
        k=0 # Assign start-index variable, so we can start next loop from index where last item is removed
        stop = 1 # Assign dummy variable to determine if we have removed something and can stop the current list index
        # (stop == 0 means that we have not stopped in an entire run-through, and can then jump to next index)
        for j in range(99999999):
            if stop == 0: # If we have not stopped in the previous run-through of current index there are no inconsistensies, we can jump to next index
                break
            stop = 0 # Set stop variable
            for i in range(k,len(dates_df_train[l]),input_size): # Jump 12 each step, start at k so we dont have to start from beginning when removing
                try:
                    if dates_df_train[l].iloc[i]["tic"]!=dates_df_train[l].
                    ->iloc[i+input_size-1]["tic"]: # If the firm 11 places in front is different we dont have chunk of 12
                        dates_df_train[l] = dates_df_train[l].drop(i).reset_index(drop=True) # Remove current index and reset index
                        k = i # Set start-index, since the next inconsistency will always come after the one we just removed
                        stop = 1 # Since we have removed datapoint, we have stopped, and thus it is not time to break out of loop since there might be more inconsistencies
                        break
                except:
                    dates_df_train[l] = dates_df_train[l].drop(i).reset_index(drop=True) # We will end up in except statement near the end if there is inconsistency. Remove this
                    k = i # Set start-index, since the next inconsistency will always come after the one we just removed
                    stop = 1 # Since we have removed datapoint, we have stopped, and thus it is not time to break out of loop since there might be more inconsistencies
    return dates_df_train
```

```python
def remove_input_size_errors_test(dates_df_test):
    # Remember that this list is a level deeper, so we need to keep track of both year and month
    year = 0
    month = 0
    for l in tqdm_notebook(range(len(dates_df_test)*12)): # The total number of indexes is the amount of years multiplied with 12
        k=0 # Assign start-index variable, so we can start next loop from index where last item is removed
        stop = 1 # Assign dummy variable to determine if we have removed something and can stop the current list index
        for j in range(9999999):
            if stop == 0: # If we have not stopped in the previous run-through of current index there are no inconsitensies, we can jump to next index
                break
            stop = 0 # Set stop variable
            for i in range(k,len(dates_df_test[year][month]),input_size): # Jump 12 each step, start at k so we dont have to start from beginning when removing
                try:
                    if dates_df_test[year][month].iloc[i]["tic"]!=dates_df_test[year][month].
                    ->iloc[i+input_size-1]["tic"]: # If the firm 11 places in front is different we dont have chunk of 12
                        dates_df_test[year][month] = dates_df_test[year][month].drop(i).reset_index(drop=True) # Remove current index and reset index
                        k = i # Set start-index, since the next inconsistency will always come after the one we just removed
                        stop = 1 # Since we have removed datapoint, we have stopped, and thus it is not time to break out of loop since there might be more inconsistencies
                        break
                except:
                    dates_df_test[year][month] = dates_df_test[year][month].drop(i).reset_index(drop=True) # We will end up in except statement near the end if there is inconsistency. Remove this
                    k = i # Set start-index, since the next inconsistency will always come after the one we just removed
                    stop = 1 # Since we have removed datapoint, we have stopped, and thus it is not time to break out of loop since there might be more inconsistencies
        month = (month+1) % 12 # Update month (every time we reach 12 it will reset to 0 and begin a new year)
        year = int(np.floor((l+1)/12)) # Update year, only change when we reach 12th, 24th, ... index
    return dates_df_test

dates_df_train = remove_input_size_errors_train(dates_df_train)
dates_df_test_dict = remove_input_size_errors_test(dates_df_test_dict)
dates_df_test_pf = remove_input_size_errors_test(dates_df_test_pf)
```

The next functions split datasets into features and target, but also keeps individual firms separated. We want the data in a way where 12 months of features correspond to the last month's target. Example: We save features for one firm from Jan to Dec, and save the target for Dec.

Again, bottom function is a level deeper.

```python
def split(dictionary):
    # Create dictionaries that will contain features and target
    X = {}
    y = {}
    for i in tqdm_notebook(range(len(dictionary))):
        # Create temporary lists that will contain dataframes containg chunks of 12 for each
        ↪  individual firm, so 1 dataframe per firm
        save_X = []
        save_y = []
        try:
            for j in range(len(dictionary[i])):
                # If next firm is different, we need to save the features from previous 11
                ↪  months and from current month + target from current month
                if dictionary[i]["tic"].iloc[j] != dictionary[i]["tic"].iloc[j+1]:
                    save_X.append(dictionary[i].iloc[j-input_size+1:j+1,2:-1]) # Save
                    ↪  features
                    save_y.append(dictionary[i].iloc[j,-1]) # Save target

        except: # We end up in the except statement in the end
            save_X.append(dictionary[i].iloc[j-input_size+1:j+1,2:-1]) # Save features
            save_y.append(dictionary[i].iloc[j,-1]) # Save target
        #Save temporary lists in dictionary
        X[i] = save_X
        y[i] = save_y
    return X,y

def split_test(test_dict):
    # Create dictionaries that will contain features and target
    X_temp,y_temp = {},{}
    for k in range(len(test_dict)):
        X_,y_ = split(test_dict[k]) # Use above function for every year of test set
        X_temp[k] = X_ # Save features for each year in dictionary
        y_temp[k] = y_ # Save target for each year in dictionary
    return X_temp,y_temp

X_train, y_train = split(dates_df_train)
X_test, y_test = split_test(dates_df_test_dict)
```

We are now ready to define and run the model.

```python
# Create RNN model that we use in project
def create_model_rnn(neurons,learning_rate,drop_out,hidden):
    early = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=50) # Stop if no
    ↪  progress in # epochs
    checkpoint = tf.keras.callbacks.ModelCheckpoint("weights.best.hdf5",monitor='val_loss',
    ↪  verbose=0, save_best_only=True, mode='min')
    callbacks_list = [early,checkpoint]
    model = Sequential()
    model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
    ↪  n_features), return_sequences=True))
    model.add(Dropout(rate=drop_out))

    # Create functionality for using different amounts of hidden layers used in grid search
    if hidden == 1:
        model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
        ↪  n_features), return_sequences=True))
    elif  hidden == 2:
        model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
        ↪  n_features), return_sequences=True))
        model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
        ↪  n_features), return_sequences=True))
    elif  hidden == 3:
        model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
        ↪  n_features), return_sequences=True))
        model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
        ↪  n_features), return_sequences=True))
        model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
        ↪  n_features), return_sequences=True))
    elif  hidden == 4:
        model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
        ↪  n_features), return_sequences=True))
        model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
        ↪  n_features), return_sequences=True))
        model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
        ↪  n_features), return_sequences=True))
        model.add(SimpleRNN(units = neurons, activation = 'relu', input_shape=(input_size,
        ↪  n_features), return_sequences=True))

    model.add(SimpleRNN(units = neurons, activation = 'relu', return_sequences=False ))
    model.add(Dense(units = 1)) #Linear output layer
    opt = optimizers.Adam(lr=learning_rate, clipnorm=1.)
    model.compile(optimizer = opt, loss = "mse")
    return model,callbacks_list
```

Below, we do the grid search. We run a model for every combination of hyperparameters given above and sort the different models on their validation mse.

```
#Hyperparameters for small grid
neurons = [50]; epochs = [50]; learning_rate = [0.0005,0.001]
drop_out = [0.05,0.01];batch_size = [32,64]; hidden = [1,2]
val_size = 0.2; loss = "mse"
def find_optimal_model(X_train,y_train,starting_point,end_point,
        ->epochs,neurons,learning_rate,batch_size,hidden,drop_out):
    optimal = []
    for i in tqdm_notebook(range(starting_point,end_point+1)):
        temp = []
        for j in range(len(neurons)):
            for q in range(len(epochs)):
                for k in range(len(learning_rate)):
                    for p in range(len(drop_out)):
                        for l in range(len(batch_size)):
                            for h in range(len(hidden)):
                                # Create model using function above
                                model,callbacks_list =
                                ↪  create_model_rnn(neurons[j],learning_rate[k],drop_out[p],
                                    ->hidden[h])
                                # Fit model on training data, save loss history
                                hist = model.fit(np.array(X_train[i]),np.array(y_train[i]),
                                    ->epochs=epochs[q],
                                        ->verbose=verbose,callbacks=callbacks_list,
                                    ->validation_split=0.2,batch_size = batch_size[l])
                                # Predict data using train as input to calculate train MSE
                                pred_train = model.predict(np.array(X_train[i]))
                                # Predict data using validation data as input to calculate
                                ↪  validation MSE
                                pred_val = model.predict(np.array(X_train[i])
                                    ->[int(len(np.array(X_train[i]))*0.8):])
                                # Save dataframe of different MSEs, hyperparameters used,
                                ↪  models, and loss results
                                temp.append([mean_squared_error(np.array(y_train[i]),
                                    ->pred_train),mean_squared_error(np.array(y_train[i])
                                    ->[int(len(y_train[i])*0.8):],pred_val),neurons[j],
                                    ->epochs[q],learning_rate[k],drop_out[p],batch_size[l],
                                    ->hidden[h],model,hist.history])
        df_temp = pd.DataFrame(data = temp,columns = ("train mse","val
        ↪  mse","neurons","epochs","learning_rate","dropout","batch size","hidden
        ↪  layers","model_save","loss_hist"))
        optimal.append(df_temp)
    return optimal, starting_point
```

run_multiple function makes it possible to run big grid on first month and then use optimal hyperparameters on next 2 months.

```
def run_multiple(X_train,y_train,starting_point,end_point,epochs,neurons,
    ->learning_rate,batch_size,hidden,drop_out):
    optimal_1, index_start = find_optimal_model(X_train,y_train,
        ->starting_point,starting_point,epochs,neurons,learning_rate,batch_size,
        ->hidden,drop_out)
    temp_hyper = optimal_1[0][optimal_1[0]["val mse"]==(optimal_1[0]["val mse"].min())]
    optimal_2, start = find_optimal_model(X_train,y_train,starting_point+1,
        ->end_point,[temp_hyper.iloc[0]["epochs"]],[temp_hyper.iloc[0]["neurons"]],
        ->[temp_hyper.iloc[0]["learning_rate"]],[temp_hyper.iloc[0]["batch
        ↪  size"]],[temp_hyper.iloc[0]["hidden layers"]],[temp_hyper.iloc[0]["dropout"]])
    return optimal_1+optimal_2,index_start, end_point
# Run above functions to find models, specify start and end index
start = ...
ending = ...
optimal1,index_start,index_end =
↪  run_multiple(X_train,y_train,start,start+2,epochs,neurons,learning_rate,
    ->batch_size,hidden,drop_out)
optimal2,index_start,slut =
↪  run_multiple(X_train,y_train,index_end+1,ending,epochs,neurons,learning_rate,
    ->batch_size,hidden,drop_out)
optimal = optimal1+optimal2
# Find every optimal model and save in dataframe along with hyperparameters used
def choose_optimal(optimal,index_start):
    df = pd.DataFrame()
    year = int(np.floor(index_start/12))
    month = index_start % 12
    for i in range(len(optimal)):
        temp = optimal[i][optimal[i]["val mse"]==(optimal[i]["val mse"].min())]
        df = df.append(temp)
        print(year,month)
        month = month+1
    df = df.reset_index(drop=True)
    return df
optimal_df = choose_optimal(optimal,start)
```

We are now ready for the portfolio stuff.

First function is what goes through every month and manages portfolios.

```python
def optimal_hyper(optimal_df,X_test,y_test,dates_df_test_pf, frequency):
    df = pd.DataFrame(columns = ("ls_ret_eq","ls_ret_mc","bh_ret_eq","bh_ret_mc")) # Create
    ↪  df that will save returns from all portfolios each month
    # We need to keep track of the current year and month that we are running through. Start
    ↪  is year 0 month 0.
    year = -1 # make sure modulus starts at right point, year will be increasesed to 0. This
    ↪  is done beacuse when i=0 we will go into year increase
    month = 0

    rebal = 1 # Initialize paramater that keeps track of when we are in a month where we can
    ↪  rebalance

    bh_ret_eq,bh_ret_mc = 0,0 # we have no return from buy/hold in first period
    bh_value_eq,bh_value_mc = 100,100 # Start with £100 (We actually ended up not using the
    ↪  value)
    bh_pf_eq,bh_pf_mc = pd.DataFrame(),pd.DataFrame() # we have no portfolios in first
    ↪  period
    portfolio_stocks = {} # Initialize dictionary that will hold all stocks we use for every
    ↪  month
    returns_list = [] # Initialize list that will save returns from every portfolio for
    ↪  every month

    save_i = [] # ALlocate storage to save indexes of rebalance points
    for i in range(len(optimal_df)):
        # First if-statement keeps track of when we can rebalance
        if i %frequency !=0: # If freq is 1 then we rebalance every month. If freq is 3, we
        ↪  can rebalance every third month and so on
            bh_ret_eq = 999 # If we are in a month where we can NOT rebalance, we just set
            ↪  the return to a large value, so we do not enter rebalance in make_pf func
            bh_ret_mc = 999
            rebal = 0 # Set rebal = 0 when we can not rebalance
        elif i%frequency == 0: # In a month we can rebalance, save the index, so we can plot
        ↪  it
            save_i.append(i)

        temp = optimal_df.iloc[i] # Save current month's predictions etc. in temp value

        # Update year every 12th index, at the same time set month to 0 for the new year
        if i %12 == 0:
            year = year+1
            month = 0
        # Run select_stocks to save tickers, predicitons, true returns, and market cap for
        ↪  each month
        portfolio_stocks[i] =
        ↪  select_stocks(temp["pred"],np.array(y_test[year][month]),dates_df_test_pf,year,
            ->month)
        # Run make_pf to create/update/manage portfolios for every month - function returns:
        # Returns from both long/short pf
        # Stocks in both buy/hold pf, returns from both buy/hold, value of both buy/hold
        ls_ret_eq, ls_ret_mc, bh_pf_eq, bh_pf_mc, bh_ret_eq, bh_ret_mc, bh_value_eq,
        ↪  bh_value_mc =
        ↪  make_pf(portfolio_stocks[i],bh_ret_eq,bh_ret_mc,bh_value_eq,bh_value_mc,
            ->bh_pf_eq,bh_pf_mc,Nasdaq_avg_ret[0],rebal )

        df.loc[len(df)] = [ls_ret_eq,ls_ret_mc,bh_ret_eq,bh_ret_mc] # Save returns in df

        returns_list.append([ls_ret_eq,ls_ret_mc,bh_ret_eq,bh_ret_mc]) # Save returns in
        ↪  list (we ended up not using this)

        month = month+1 #increase 1 month
        rebal = 1 # Reset rebal for next month
    df = df.reset_index(drop=True)
    return df, portfolio_stocks, returns_list, save_i


    def select_stocks(pred_ret,true_ret,dates_df_test_pf,year,month):
            # Function gathers tickers, predicted returns, true returns, and market cap for
            ↪  all stocks
            tic_cap = pd.DataFrame()
            for l in range(2,len(dates_df_test_pf[year][month]),input_size): # Get tickers
            ↪  from dates_df_test_pf, but only need 1 instance per firm
                tic_cap = tic_cap.append(dates_df_test_pf[year][month][l:l+1])

            df = pd.DataFrame(columns = ("tic","pred","true","market cap"))
            df["tic"] = tic_cap["tic"]
            df["pred"] = np.exp(pred_ret)-1 #Transform log-returns to normal returns
            df["true"] = np.exp(true_ret)-1 #Transform log-returns to normal returns
            df["market cap"] = tic_cap["mkt cap"]
            df = df.sort_values("pred",ascending=False) # Sort everything according to
            ↪  predicted returns - highest to lowest
    return df
```

```python
def make_pf(df,bh_ret_eq,bh_ret_mc,bh_value_eq,bh_value_mc,bh_pf_eq,bh_pf_mc,Nasdaq_avg_ret,rebal):
    # Function should create 2 portfolios: Long/short & buy/hold

    '''_______________ long/short _______________'''
    # First create long/short portfolios that go long in top 10% performing stocks and short in bottom 10% performing stocks

    ls_pf = df.copy().drop(df.index[int(len(df)*0.1):int(len(df)*0.9)]).reset_index(drop=True) # Create df 20% of the length of input stocks

    if len(ls_pf) % 2 != 0: # For zero-net investment, we make sure that we long and short same amount of stocks
        ls_pf = ls_pf.drop(np.floor(len(ls_pf)/2)).reset_index(drop=True) # Thus, if we have odd amount of stocks, remove middle index

    # Equal weight
    ls_pf["equal"] = (ls_pf["true"])*1/(len(ls_pf)/2) # Multiply both long and short positions with 1/(length of long/short)
    ls_pf["equal"].iloc[len(ls_pf)//2:] = ls_pf["equal"].iloc[len(ls_pf)//2:]*(-1) # Multiply short positions with minus 1 (get opposite sign returns)
    ls_ret_eq = np.sum(ls_pf["equal"]) # The return of the portfolio will be the sum of the returns

    # Market cap weight
    # For the market-cap-weighted do the same as above, but this time multiply by weights: mkt_cap/(total mkt_cap of long/short positions)
    ls_pf["mkt_cap"] = (ls_pf["true"])
    ls_pf["mkt_cap"].iloc[:len(ls_pf)//2] = ls_pf["mkt_cap"].iloc[:len(ls_pf)//2]*1*ls_pf["market cap"].iloc[:len(ls_pf)//2]/sum(ls_pf["market cap"].iloc[:len(ls_pf)//2])
    ls_pf["mkt_cap"].iloc[len(ls_pf)//2:] = ls_pf["mkt_cap"].iloc[len(ls_pf)//2:]*(-1)*ls_pf["market cap"].iloc[len(ls_pf)//2:]/sum(ls_pf["market cap"].iloc[len(ls_pf)//2:])
    ls_ret_mc = np.sum(ls_pf["mkt_cap"]) # The return of the portfolio will be the sum of the returns


    '''_______________ buy/hold _______________'''
    # Next create buy/hold portfolios that buy stocks with positive predictions and holds, and goes cash when market goes down

    # Equal weight
    # Check if we should rebalance or not depending on the return from previous period
    if bh_ret_eq >= Nasdaq_avg_ret:
        if bh_pf_eq.empty: # If we are holding cash, the return from this month is 0
            bh_ret_eq = 0
        elif len(df[df["pred"]>0])<1 and rebal == 1: # If we should not rebalance, check if next month will go down
            # If yes, and we are able to rebalance, go cash to avoid next month's crash
            bh_ret_eq = 0
        else:
            # If we are in a situation where we should not rebalance, save current stocks in the portfolio
            tics = np.unique(df["tic"])
            bh_pf_eq = bh_pf_eq[bh_pf_eq["tic"].isin(tics)].sort_values("tic")
            bh_pf_eq["true"] = df[df["tic"].isin(bh_pf_eq["tic"].unique())].sort_values("tic")["true"].
            ->tolist()
            bh_pf_eq["equal"] = (bh_pf_eq["true"]+1)*bh_value_eq/len(bh_pf_eq) # Calculate weighted return of each stock in pf (equal)
            bh_ret_eq = (sum(bh_pf_eq["equal"])/bh_value_eq)-1 # Find total return for month
            bh_value_eq = sum(bh_pf_eq["equal"]) # Update total pf value

    # Check if returns are below threshold so we should rebalance (If we are in a month where we can NOT rebalance, we will never enter here)
    elif bh_ret_eq < Nasdaq_avg_ret:
        bh_pf_eq = df[df["pred"]>0].copy() # Rebalance by buying every stock with positive prediction
        if bh_pf_eq.empty: # If all predictions are negative, we go cash
            bh_ret_eq = 0
        elif len(bh_pf_eq)<1: # Same as above, but can be changed if we want more positive predictions before we buy
            bh_ret_eq = 0
        else:
            bh_pf_eq["equal"] = (bh_pf_eq["true"]+1)*bh_value_eq/len(bh_pf_eq) # Calculate weighted return of each stock in pf (equal)
            bh_ret_eq = (sum(bh_pf_eq["equal"])/bh_value_eq)-1 # Find total return for month
            bh_value_eq = sum(bh_pf_eq["equal"]) # Update total pf value


    # Market cap weight
    # Check if we should rebalance or not depending on the return from previous period
    if bh_ret_mc >= Nasdaq_avg_ret:
        if bh_pf_mc.empty: # If we are holding cash, the return from this month is 0
            bh_ret_mc = 0
        elif len(df[df["pred"]>0])<1 and rebal == 1: # If we should not rebalance, check if next month will go down
            # If yes, and we are able to rebalance, go cash to avoid next month's crash
            bh_ret_mc = 0
        else:
            # If we are in a situation where we should not rebalance, save current stocks in the portfolio
            tics = np.unique(df["tic"])
            bh_pf_mc = bh_pf_mc[bh_pf_mc["tic"].isin(tics)].sort_values("tic")
            bh_pf_mc["true"] = df[df["tic"].isin(bh_pf_mc["tic"].unique())].sort_values("tic")["true"].
            ->tolist()
            bh_pf_mc["mkt_cap_pf"] = (bh_pf_mc["true"]+1)*bh_value_mc/len(bh_pf_mc) # Calculate weighted return of each stock in pf (market-cap-weighted)
            bh_ret_mc = (sum(bh_pf_mc["mkt_cap_pf"])/bh_value_mc)-1 # Find total return for month
            bh_value_mc = sum(bh_pf_mc["mkt_cap_pf"]) # Update total pf value

    # Check if returns are below threshold so we should rebalance (If we are in a month where we can NOT rebalance, we will never enter here)
    elif bh_ret_mc < Nasdaq_avg_ret:
        bh_pf_mc = df[df["pred"]>0].copy() # Rebalance by buying every stock with positive prediction
        if bh_pf_mc.empty: # If all predictions are negative, we go cash
            bh_ret_mc = 0
        elif len(bh_pf_mc)<1: # Same as above, but can be changed if we want more positive
        ↪  predictions before we buy
                bh_ret_mc = 0
        else:
            bh_pf_mc["mkt_cap_pf"] = (bh_pf_mc["true"]+1)*bh_value_mc*bh_pf_mc["market
            ↪  cap"]/sum(bh_pf_mc["market cap"]) # Calculate weighted return of each stock
            ↪  in pf (market-cap-weighted)
            bh_ret_mc = (sum(bh_pf_mc["mkt_cap_pf"])/bh_value_mc)-1 # Find total return for
            ↪  month
            bh_value_mc = sum(bh_pf_mc["mkt_cap_pf"]) # Update total pf value

    # return returns from all portfolios, stocks in buy/hold and value of buy/hold
    return ls_ret_eq, ls_ret_mc, bh_pf_eq, bh_pf_mc, bh_ret_eq, bh_ret_mc, bh_value_eq,
    ↪  bh_value_mc


# Create function that can make table of returns, volatility and Sharpe ratio for period
def df_results(pred_period,optimal_new):
    rf = pd.read_csv("DGS10.csv") # Get risk-free rate
    # Find Nasdaq and rf for the period we are looking at
    if pred_period == 0:
        plot_nas = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2005-01-31") &
        ↪  (Nasdaq100_index["DATE"]<"2012-01-01")]["NASDAQ100"].tolist()
        rf = rf[(rf["DATE"]>"2005")&(rf["DATE"]<"2012")]["DGS10"].mean()

    elif pred_period == 1:
        plot_nas = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2012-01-31") &
        ↪  (Nasdaq100_index["DATE"]<"2019-01-01")]["NASDAQ100"].tolist()
        rf = rf[(rf["DATE"]>"2012")&(rf["DATE"]<"2019")]["DGS10"].mean()

    elif pred_period == 2:
        plot_nas = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2019-01-31") &
        ↪  (Nasdaq100_index["DATE"]<"2023-01-01")]["NASDAQ100"].tolist()
        rf = rf[(rf["DATE"]>"2019")&(rf["DATE"]<"2023")]["DGS10"].mean()
    # The data has "." in a lot of places, that we remove
    for x in plot_nas[:]:
        if x == '.':
            plot_nas.remove(x)
    plot_nas = [float(x) for x in plot_nas] # The dataformat is also strings, so we convert
    ↪  to float
    # Calculate Nasdaq returns
    ret_nas = []
    for i in range(1,len(plot_nas)):
        ret_nas.append(plot_nas[i]/plot_nas[i-1]-1)
    Nasdaq_ret = np.cumsum(ret_nas)[-1] # Find total Nasdaq return in period
    Nasdaq_avg_ret = Nasdaq_ret/(len(optimal_new)/12) # Find the average yearly Nasdaq
    ↪  return
    vol = np.std(ret_nas)*np.sqrt(252) # Find Nasdaq yearly volatility
    SR = (Nasdaq_avg_ret-rf)/vol # Calculate Nasdaq Sharpe ratio
    data = np.array([Nasdaq_ret,Nasdaq_avg_ret,vol,SR]) # Save Nasdaq data and put in df
    df_tabular = pd.DataFrame(data=data,columns=["Nasdaq 100"],index=["Total return in
    ↪  period","Average annual return","Annual volatility","Sharpe ratio"])

    # Allocate storage for buy/hold total return and volatility
    tot_ret_bh = []
    ann_vol_bh = []
    # Allocate storage for long/short total return and volatility
    tot_ret_ls = []
    ann_vol_ls = []

    # Calculate buy/hold total return, average annual return, annual volatility, and Sharpe
    ↪  ratio
    for col in optimal_new.columns[2:]:
        tot_ret_bh.append(np.cumsum(optimal_new[col]).tolist()[-1])
        avg_ann_ret_bh = [x/(len(optimal_new)/12) for x in tot_ret_bh]
        ann_vol_bh.append(np.std(optimal_new[col])*np.sqrt(12))
    SR_bh = (np.array(avg_ann_ret_bh)-rf)/ann_vol_bh

    # Calculate long/short total return, average annual return, annual volatility, and Sharpe
    ↪  ratio
    for col in optimal_new.columns[:2]:
        tot_ret_ls.append(np.cumsum(optimal_new[col]).tolist()[-1])
        avg_ann_ret_ls = [x/(len(optimal_new)/12) for x in tot_ret_ls]
        ann_vol_ls.append(np.std(optimal_new[col])*np.sqrt(12))
    SR_ls = (np.array(avg_ann_ret_ls)-rf)/ann_vol_ls


    df_tabular["Buy/hold eq"] = [tot_ret_bh[0],avg_ann_ret_bh[0],ann_vol_bh[0],SR_bh[0]]
    df_tabular["Buy/hold mc"] = [tot_ret_bh[1],avg_ann_ret_bh[1],ann_vol_bh[1],SR_bh[1]]
    df_tabular["Long/short eq"] = [tot_ret_ls[0],avg_ann_ret_ls[0],ann_vol_ls[0],SR_ls[0]]
    df_tabular["Long/short mc"] = [tot_ret_ls[1],avg_ann_ret_ls[1],ann_vol_ls[1],SR_ls[1]]

    return df_tabular
df_results(0,optimal_new)
```

```
# For table with yearly long/short returns + Nasdaq

a = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2005-01-31") &
↪  (Nasdaq100_index["DATE"]<"2012-01-01")]
# a = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2012-01-31") &
↪  (Nasdaq100_index["DATE"]<"2019-01-01")]
# a = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2019-01-31") &
↪  (Nasdaq100_index["DATE"]<"2023-01-01")]
for i in range(len(a)):
    if a["NASDAQ100"].iloc[i] == ".":
        a["NASDAQ100"].iloc[i] = a["NASDAQ100"].iloc[i-1]
    else:
        a["NASDAQ100"].iloc[i] = float(a["NASDAQ100"].iloc[i])
a["return"] = a["NASDAQ100"]/a["NASDAQ100"].shift(1)-1
a = a.dropna()

years = np.arange(2012,2019) # Change for period
total_sum = []
for year in years:
    total_sum.append(sum(a[(a["DATE"]>str(year-1)+"12-31") &
    ↪  (a["DATE"]<str(year+1))]["return"].tolist()))

ls_df = pd.DataFrame(index=years)
ls_eq_cum_ret = []
ls_mc_cum_ret = []
for i in range(11,len(optimal_new)+11,12):
    ls_eq_cum_ret.append(optimal_new["ls_ret_eq"].loc[i-11:i].sum())
    ls_mc_cum_ret.append(optimal_new["ls_ret_mc"].loc[i-11:i].sum())
ls_df["Nasdaq 100"] = total_sum
ls_df["Long/short eq"] = ls_eq_cum_ret
ls_df["Long/short mc"] = ls_mc_cum_ret
ls_df.loc["Total"] = [sum(total_sum),sum(ls_eq_cum_ret),sum(ls_mc_cum_ret)]


In the following function, we plot the cumulative returns for nasdaq and for buy/hold with 3
↪  different rebalancing frequencies.
```

In the following function, we plot the cumulative returns for nasdaq and for buy/hold with 3 different rebalancing frequencies.

```python
def compare_return_with_nasdaq(optimal_df,Nasdaq100_index,pred_period,pf_type):
    rebalance = 1 # Determine rebalancing frequency
    for k in range(3):
        if k==1:
            rebalance = 3
        elif k==2:
            rebalance = 6
        optimal_new, portfolio_stocks, returns_list, save_i =
        ↪  optimal_hyper(optimal_df,X_test,y_test,dates_df_test_pf,rebalance)

        # Determine period we are looking at, and find correspodning Nasdaq dates
        if pred_period == 0:
            date_range = pd.date_range(start='2005-01-01', end='2012-01-31', freq='M')
            plot_nas = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2005-01-31") &
            ↪  (Nasdaq100_index["DATE"]<"2012-02-01")]["NASDAQ100"].tolist()
            X_axis = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2005-01-31") &
            ↪  (Nasdaq100_index["DATE"]<"2012-01-31")]["DATE"].tolist()
        elif pred_period == 1:
            date_range = pd.date_range(start='2012-01-01', end='2019-01-31', freq='M')
            plot_nas = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2012-01-31") &
            ↪  (Nasdaq100_index["DATE"]<"2019-02-01")]["NASDAQ100"].tolist()
            X_axis = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2012-01-31") &
            ↪  (Nasdaq100_index["DATE"]<"2019-01-31")]["DATE"].tolist()
        elif pred_period == 2:
            date_range = pd.date_range(start='2019-01-01', end='2023-02-01', freq='M')
            plot_nas = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2019-01-31") &
            ↪  (Nasdaq100_index["DATE"]<"2023-02-01")]["NASDAQ100"].tolist()
            X_axis = Nasdaq100_index[(Nasdaq100_index["DATE"]>="2019-01-31") &
            ↪  (Nasdaq100_index["DATE"]<"2023-01-31")]["DATE"].tolist()
        else:
            print("WRONG INPUT PRED PERIOD")

        ''' PREPARE NASDAQ 100 PLOT '''
        index_list = []
        i = 0
        for x in plot_nas[:]:
            if x == '.':
                plot_nas.remove(x)
                index_list.append(i)
            i = i+1
        plot_nas = [float(x) for x in plot_nas]
        cum_ret_nas = []
        for i in range(1,len(plot_nas)):
            cum_ret_nas.append(plot_nas[i]/plot_nas[i-1]-1)
        cum_ret_nas = np.cumsum(cum_ret_nas) # Calculate cumulative returns for Nasdaq

        # Prepare x-axis for Nasdaq plot
        X_axis = [x for i, x in enumerate(X_axis) if i not in index_list]
        X_axis = [datetime.datetime.strptime(date_str, "%Y-%m-%d").date() for date_str in
        ↪  X_axis]

        eq = [0]
        mc = [0]

        ''' PREPARE RETURN PLOT'''
        if pf_type == "ls":
            eq = eq + (optimal_new["ls_ret_eq"]).tolist() # Collect lists
            mc = mc + (optimal_new["ls_ret_mc"]).tolist()
        elif pf_type == "bh":
            eq = eq + (optimal_new["bh_ret_eq"]).tolist() # Collect lists
            mc = mc + (optimal_new["bh_ret_mc"]).tolist()
        else:
            print("WRONG INPUT PF TYPE")

        plot_eq = np.cumsum(eq) # Calculate cumulative returns for equal buy/hold
        plot_mc = np.cumsum(mc) # Calculate cumulative returns for mkt cap buy/hold

        # Add recession bars to plot
        if pred_period == 0:
            rec_3 = [13847,14395] #rec_3: 2007-12-01, 2009-06-01
            plt.fill_between(rec_3, -1.1, 5.1, facecolor=(0,0,0,.05), edgecolor=(0,0,0,.2))
            plt.ylim(-0.35,2.15)
        elif pred_period == 2:
            rec_4 = [18292,18352] #rec_4: 2020-02-01, 2020-04-01
            plt.fill_between(rec_4, -0.25, 1.51, facecolor=(0,0,0,.05),
            ↪  edgecolor=(0,0,0,.2))
            plt.ylim(-0.2,1.35)
        x_pf = np.array(date_range)
        plt.style.use('default')

        # Make sure we do not plot rebalnce points if we can rebalance often or are looking
        ↪  at long/short
        if len(save_i)>40 or pf_type == "ls":
            marker_on = []
        else:
            marker_on = save_i

        # Plot
        if k==0:
            plt.plot(X_axis,cum_ret_nas,label = "Nasdaq 100 index")
            plt.plot(x_pf,plot_eq,label = "Equal reb 1",color = "dimgrey")
            plt.plot(x_pf,plot_mc,"--",label = "Market cap reb 1",color = "black")

        elif k==1:
            plt.plot(x_pf,plot_eq,"-o",markevery=marker_on,label = "Equal reb 3",color =
            ↪  "forestgreen")
            plt.plot(x_pf,plot_mc,"--*",markevery=marker_on,label = "Market cap reb 3",color
            ↪  = "limegreen")
        elif k==2:
            plt.plot(x_pf,plot_eq,"-o",markevery=marker_on,label = "Equal reb 6", color =
            ↪  "tomato")
            plt.plot(x_pf,plot_mc,"--*",markevery=marker_on,label = "Market cap reb 6",color
            ↪  = "firebrick" )

        plt.xlabel("Date",fontsize = 15)
        plt.ylabel("Cumulative returns",fontsize = 15)

        plt.grid()
        plt.legend(loc = 2)

plt.figure(figsize = (15,10))
compare_return_with_nasdaq(optimal_df,Nasdaq100_index,0,"bh")


# Prepare loss plots
true_train = []
true_val = []
for j in range(100):

    temp_1 = []
    temp_2 = []
    for i in range(0,len(optimal_df),3):
        try:
            temp_1.append(optimal_df["loss_hist"][i]["loss"][j])
            temp_2.append(optimal_df["loss_hist"][i]["val_loss"][j])
        except:
            None
    true_train.append(np.mean(temp_1))
    true_val.append(np.mean(temp_2))
plt.figure(figsize=(15,10))
plt.plot(true_train[1:],label = "training loss")
plt.plot(true_val[1:],label = "validation loss")
plt.ylim(0.035,0.062)
x_lab = [1,10,20,30,40,50]
x_ticks = [0,10,20,30,40,48]
plt.xticks(ticks=x_ticks, labels=x_lab)

plt.xlabel("Epochs",fontsize = 15)
plt.ylabel("Loss",fontsize = 15)
plt.grid()
plt.legend(loc=1,fontsize = 15)
```