DIPLOMARBEIT

A Deep Learning Approach for Analyzing the Limit Order Book

ausgeführt am

Institut für Stochastik und Wirtschaftsmathematik
TU Wien

unter der Anleitung von

Univ.Prof. Dipl.-Math. Dr.rer.nat. Thorsten Rheinländer

durch

David Hirnschall, BSc
Matrikelnummer: 01427610

18. August 2020

Kurzfassung

In den letzten Jahrzehnten haben sich die Finanzmärkte aufgrund der enormen, heutzutage verfügbaren, Datenmenge grundlegend verändert. In dieser Arbeit stellen wir einen rein datengetriebenen Ansatz ohne zugrunde liegende Annahmen, wie etwa Preisdynamiken, für die Analyse der Märkte vor. Wir verwenden modernste Techniken des maschinellen Lernens, um den Informationsgehalt von Limit Order Büchern (LOB) zu untersuchen, indem wir die Vorhersage von Preisbewegungen sowie Volatilitäten, zwei der wichtigsten Fragestellungen für Investoren, behandeln.
Wir beginnen mit einer detaillierten Einführung in die Theorie neuronaler Netze, wo wir nicht nur grundlegende Architekturen, insbesondere deep feedforward neuronale Netze, und deren Trainingsalgorithmus vorstellen, sondern uns auch auf Optimierungs- und Generalisierungstechniken konzentrieren. Darüber hinaus präsentieren wir eine mathematisch exakte Beschreibung eines Limit Order Buches sowie seiner tatsächlichen Datenstruktur, gefolgt von einem unverzerrten Schätzer der realisierten Volatilität unter Verwendung von verrauschten Hochfrequenzdaten. Dieser wird TSRV (Two Scales Realized Volatility Estimator) genannt.
Schließlich zeigen wir empirische Ergebnisse für vier verschiedene Aktien. Die verwendeten Daten des Limit Order Buches von der NASDAQ, der zweitgrößten Börse weltweit, wurden vom Online-Tool LOBSTER bereitgestellt. Für jede Fragestellung extrahieren wir zunächst eine breite Palette technischer und quantitativer Merkmale aus den Daten des Limit Order Buches. Anschließend verwenden wir Methoden, wie etwa recursive feature selection und den Boruta-Algorithmus, zur Auswahl der wichtigsten Merkmale um die Trainingsgeschwindigkeit zu erhöhen und die Leistung zu verbessern. Durch die Verwendung von deep feedforward neuronalen Netzen, die auf den wichtigsten Merkmalen trainiert worden sind, können wir häufig verwendete lineare Algorithmen, wie die logistische Regression, für die Vorhersage des mittleren Preises für alle vier Aktien übertreffen. Darüber hinaus liefert unser vorgeschlagener Ansatz weitaus bessere, langfristige Volatilitätsprognosen als ARIMA Modelle. Folglich wird die Notwendigkeit einer Neukalibrierung verringert, wodurch schnellere Vorhersagen und damit potenziell vorteilhafte Indikatoren für Anleger ermöglicht werden.

Abstract

In the last few decades the financial markets have changed fundamentally because of the tremendous amount of data which is available nowadays. In this thesis we propose a purely data-driven approach without any underlying assumptions, such as price dynamics, for analyzing the markets. We therefore use state of the art machine learning techniques to investigate the information content of limit order books (LOB), by targeting two of the most important tasks for investors, namely predicting price movements and forecasting volatility.
We start by giving a detailed introduction into the theory of neural networks, where we not only present basic architectures, particular deep feedforward neural networks, and how to train them, but also focus on optimization and generalization techniques. Furthermore, we present a mathematically precise description of a limit order book as well as its actual data structure, followed by an unbiased approach to estimate realized volatility using noisy high-frequency data, referred to as two scales realized volatility estimator (TSRV).
Finally, we show empirical results for four different stocks. The used limit order book data from NASDAQ, the second largest stock exchange in the world, was provided by the online tool LOBSTER. For each task we first extract a wide range of technical and quantitative features from basic limit order book data. Afterwards, we use feature selection methods, such as recursive feature selection and the Boruta algorithm to increase training speed and improve performance. By using deep feedforward neural networks, trained on the most important features, we are able to outperform commonly used linear algorithms for mid-price prediction such as multiclass logistic regression for all four stocks. Additionally, our proposed approach yields better long term volatility forecasts than ARIMA models. Consequently, it reduces the necessity of re-calibration, which yields faster predictions and therefore potentially beneficial indicators for investors.

Acknowledgment

My gratitude goes to Univ.Prof. Dipl.-Math. Dr.rer.nat. Thorsten Rheinländer for supervising this thesis and being available at all times to answer my questions.
I would like to express my sincere thanks to Sandra Trenovatz, secretary at FAM, for finding solutions for every organizational problem, but especially for always having students' backs.
Furthermore, I would like to say thank you to my family and friends, especially my mother, without whose never ending support I would not have gotten this far. Thank you for being there for me in every imaginable situation.
Last but not least, a special thanks goes my amazing girlfriend not only for her ongoing love and support but also for always motivating me to do my best.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am 18. August 2020

David Hirnschall

Contents

1. Introduction

We are drowning in information and starving for knowledge. — John Naisbitt.

With the rapid increase in the amount of data available due to modern technologies, the need for automated methods for data analysis continues to grow. We want to define the set of these methods as machine learning. The goal of machine learning is to automatically detect patterns in data and then to use these patterns to identify complex relationships or to predict future outcomes [26]. In other words we are looking for algorithms which learn from given data, even though they are not explicitly programmed to do so.
These methods, particularly deep neural networks, have been a research topic for many years and some of major statements have been discovered already 20 years ago in 1989 by Kurt Hornik, Maxwell Stinchcombe, and Halbert White [17]. Back then their usability was very limited due to the lack of computational power. With modern computers machine learning has become one of the most important subjects among researchers and companies.
Nowadays there are numerous areas of application for machine learning algorithms, such as science, marketing, natural disaster prediction, image recognition and also finance and insurance. Particularly deep learning models like deep feedforward neural networks have already found numerous applications in quantitative finance, such as price prediction or volatility forecasting.
Although neural networks can yield great results, their calculation process is hard to interpret and therefore deep learning is frequently criticized as black box with lacking fundamental theory.

1.1. Machine Learning - Terminology

Before going more into detail of deep neural networks and how to use them in finance, we give a brief introduction in important machine learning terminologies.

1.1.1. Types of machine learning

Usually machine learning algorithms are divided into three, sometimes even into four groups, namely supervised, unsupervised, reinforcement and sometimes semi-supervised learning [4].

Supervised Learning In supervised learning the data set contains out of labeled examples $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$, where $\boldsymbol{x}_i \in \mathbb{R}^d$ is referred to as a d-dimensional feature vector and represents one observation, whereas y_i is referred to as target or response variable. The

feature vector $\boldsymbol{x}_i$ contains in each dimension $j = 1, \ldots, d$ one observed value that somehow describes the target y_i. These values are called features, attributes or covariates and are donated as $\boldsymbol{x}_i^{(j)}$. The labels y_i can either be discrete, in particular an element belonging to a finite set of classes, such as $\{K_1, K_2, \ldots, K_m\}$ or $\{good, bad\}$, which yields a classification problem, or be continuous (eg. $y_i \in \mathbb{R}$), which yields a regression problem. The goal of supervised learning is to produce a model which uses $\boldsymbol{x}_i$ to predict some output that allows us to deduce y_i. More precisely, we want to solve a regression problem by approximating a function $f : f(\boldsymbol{x}_i) = y_i,\ i = 1, \ldots, N$, and calculate the conditional probability $p(y_i|\boldsymbol{x}_i)$ for a classification problem. In Section 2 we deal with both tasks in more detail.

Unsupervised Learning In unsupervised learning the data set contains unlabeled examples $D = \{\boldsymbol{x}_i\}_{i=1}^N$, where $\boldsymbol{x}_i$, again represents the d-dimensional feature vector but this time no target variable is available. The goal here is to learn the unconditional probability $p(\boldsymbol{x}_i)$ and find patterns in the data set. Some of the most used unsupervised learning algorithms are clustering, which divides the data set into groups according to interesting patterns, or dimensionality reduction techniques to reduce the number of features in a data set without losing much information and keeping or even improving the model's performance.
We do not cover unsupervised learning in this thesis.

Reinforcement Learning In reinforcement learning we are not dealing with a predefined data set but rather the machine uses available features of every state in an environment and can take actions there. Every action yields different rewards. The goal of reinforcement learning is to maximize the reward of a predefined reward function, thus it can be seen as trail and error.
Reinforcement learning is mostly used to solve problems with long term goals under sequential decision making, such as game playing or robotics (e.g. AlphaGo).
We do not use reinforcement learning in this thesis.

1.1.2. Performance

Now that we know what type of algorithms we have to use for different problems, we have to think about its performance. One of the most important aspects is how exact and complex should a model be trained? Regarding model complexity we differentiate between two main challenges, namely overfitting and underfitting.

Overfitting Fitting highly complex models often leads to overfitting, meaning a model learns too precisely from the training data. Models that make predictions based on too small details in the training data are able to divide them perfectly, but also yield poor decisions for new data, without these special details. Consequently, it is usually beneficial avoid considering every minor aspect in the input data as important, since it

is more likely to face noise than true signals. The ability to perform well on previously unknown input data is called generalization and gets discussed in Section 2.4.

Underfitting On the other hand, if the training data is not analyzed sufficiently precisely, we are talking about underfitting. The resulting, too simple, model is not able to recognize data structures, thus it does not make good predictions neither for the training data nor for new data.

A graphical illustration of under- and overfitting is given in Figure 1.1.

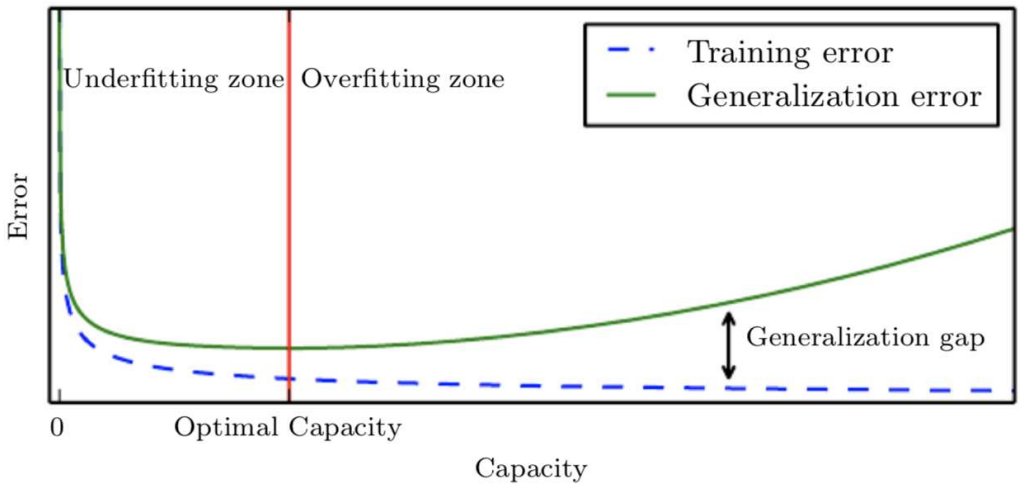


Figure 1.1: Typical relationship between a model's ability to fit a wide variety of functions, referred to as capacity, and error. [12]

As a consequence, model selection is one the most important tasks in machine learning and can make the difference between success and failure of a project. In Section 2.4.1 we are dealing with it in more detail.

1.2. Limit Order Book

The importance of algorithmic trading is growing and nowadays financial instruments such as stocks and futures are increasingly traded in electronic, order-driven markets. The heart of these modern trading systems is a double auction mechanism called limit order Book (LOB), [19], which contains all critical information of buy and sell orders for each traded stock, in particular it contains all open limit orders at each price level. A limit order is a order to buy or sell a certain amount of shares of a specific stock at a certain price. It remains in the order book until it gets executed or canceled. The buy limit orders are on the bid side and the sell limit orders on the ask side, respectively. The prices at which the stock can immediately be bought and sold, via a so called market

order, are referred to as best bid and best ask price, respectively. Market orders get executed immediately at the best level and then "walk their way through" the book for any additional shares. A graphical representation of a LOB is given in Figure 1.2.

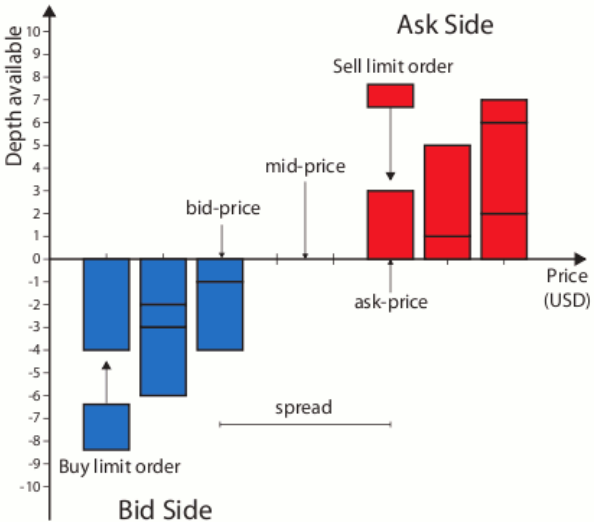


Figure 1.2: Graphical representation of a limit order book, [8]

The gap between best bid ans best ask is referred to as bid-ask-spread, and varies over time, although it is often modeled as a constant. Taking the average of best bid and best ask price yields the mid-price. It is often considered the "price" of the stock, although it is just a number at which one can not buy or sell. Therefore, theoretically it would be beneficial to model both the best bid and the best ask [34]. Nevertheless, the so called market micro-structure noise can be partially reduced by using midprices.
Since the limit order book records changes in open orders in milliseconds, it contains hundreds of prices at different levels, hence it describes the known demand and supply of a stock at every point in time. This complexity and high-dimensionallity of the order book makes modeling extremely challenging.
In this thesis we develop a purely data-driven approach to target these challenges, by using state of the art machine learning techniques. In order to capture the non-linear relationship among the different levels in the order book we use deep feedforward neural networks. For the implementation in Python (version 3.7.7) we use the Keras library (version 2.3.1) with Tensorflow (version 2.2.0) as backend.

2. Deep Learning

In this chapter we introduce deep feedforward neural networks and how they can capture non-linear relationship among input variables. We present the back propagation algorithm and investigate different optimization as well as generalization techniques.

2.1. Basic Network Architecture

Deep feedforward networks, often referred to as multilayer perceptrons, build the basis of all different kinds of extended neural network architectures. The goal of feedforward networks is to approximate an function $f : y = f(\boldsymbol{x})$, by optimazing some parameters $\boldsymbol{\theta}$ such that $\hat{f}(\boldsymbol{x}; \boldsymbol{\theta})$ yields the best approximation.
Their main components are an input layer, at least three to four (depending on the definition) hidden layers -otherwise they are called shallow networks- and one output layer. These models are called feedforward because information flows from the input layer directly though the hidden layers to the output layer without any cycles or feedbacks. Networks extended by feedbacks are referred to as recurrent networks.
In this thesis we focus on feedforward neural networks and do not cover any extensions such as recurrent neural networks or LSTM models.
Finally, they are called neural network because of the connected nodes each vector valued layer contains. Since neural networks are inspired by the human brain, the role of each node is analogous to a brain neuron. The number of neurons in a hidden layer gives the width of the network and the number of hidden layers the depth, respectively.
In this section we mainly follow [1, Section 1.2.1 and Section 1.2.2] and [12].

2.1.1. The Perceptron (computing units)

The simplest neural network is referred to as the perceptron. These neural networks contain only an input layer and one output node, the computing unit. The basic architecture can be seen in Figure 2.1.

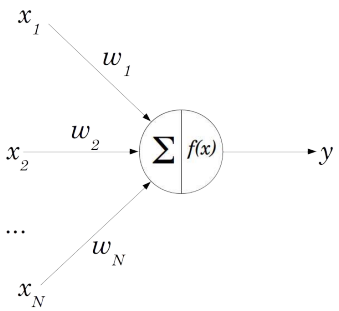


Figure 2.1: Basic architecture of a perceptron, [25]

As a basic example consider a binary classification problem with data of the form $\{(\boldsymbol{x}_i, y_i)_{i=1}^N\}$, where $\boldsymbol{x}_i = (x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(N)})$ contains N input features and $y_i \in \{-1, +1\}$ is the target variable. This example might occur in a credit card fraud detection application.
The input layer contains d nodes that send $\boldsymbol{x}_i,\ i = 1, 2, \ldots, d$ with edges of weights $\boldsymbol{w} = (w_1, w_2, \ldots, w_N)$ to the output node. There the linear function $\boldsymbol{w} \cdot \boldsymbol{x}_i = \sum_{j=1}^N w_j x_i^{(j)}$ is computed. The input layer does not perform any computations on its own and is therefore not counted as an additional layer. Subsequently, the sign function of this value is used to predict $\hat{y}_i$ out of $\boldsymbol{x_i},\ i = 1, 2, \ldots, d$ and convert the aggregated value to an class label as follows:

$$\hat{y}_i = \text{sign}(\boldsymbol{w} \cdot \boldsymbol{x}_i) = sign\left(\sum_{j=1}^{N} w_j x_i^{(j)}\right).$$

The sign function serves the role of an activation function.
In many situations a prediction is biased, i.e. there is an invariant part of the prediction. For example mean centered feature variables can yield binary class prediction from $\{-1, +1\}$, with a non-zero mean. This often occurs in situation in which the binary class distribution is highly imbalanced. In such cases we have to adjust the aforementioned approach by incorporating a additional bias variable b that captures the invariant part of the prediction. We can do that by adding a bias neuron and assigning the bias variable to its weight:

$$\hat{y}_i = \text{sign}(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) = \text{sign}\left(\sum_{j=1}^{N} w_j x_i^{(j)} + b\right).$$

Activation Functions Choosing an activation function is a critical part, when building an neural network. For a binary classification problem, such as the one above, we use the sign function. However, in different situation other functions may be better suited. For predicting a real-valued target it makes sense to use the identity activation function, which yields the same as a least squares regression. Predicting the probability of a binary class, may be achieved by the sigmoid activation function, and the resulting algorithm is the same as a logistic regression. As a matter of fact, most of the basic machine learning algorithms can be represented as simple network architectures.
Formally an activation function can be defined as follows:

Definition 2.1 (Activation Function)**.** The often non-linear activation function $\phi(\cdot)$ at a neuron n transforms the input $\boldsymbol{x}$ after multiplying it with $\boldsymbol{w}$ to $n = \phi(\boldsymbol{x} \cdot \boldsymbol{w} + b)$ with bias b.

Hence, each node actually computes two function, one linear summation and one in most cases non-linear activation function, as illustrated in Figure 2.2.

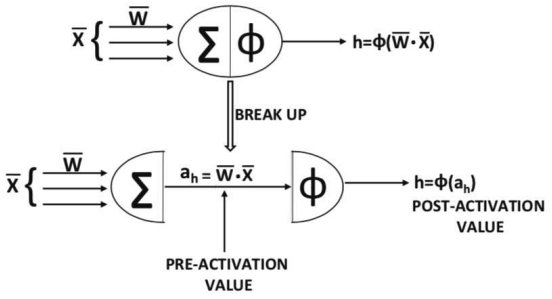


Figure 2.2: Pre- and post-activation values with a single neuron, [1]

We distinguish between the value before applying the activation function, referred to as pre-activation value, and the value computed after applying the activation function referred to as post activation value, respectively. Even though the output of a node is always the post-activation value, the pre-activation value is very important for computations, such as backward propagation discussed in Section 2.3.1.
Figure 2.3 illustrates some of the most used activation functions for a single output node.

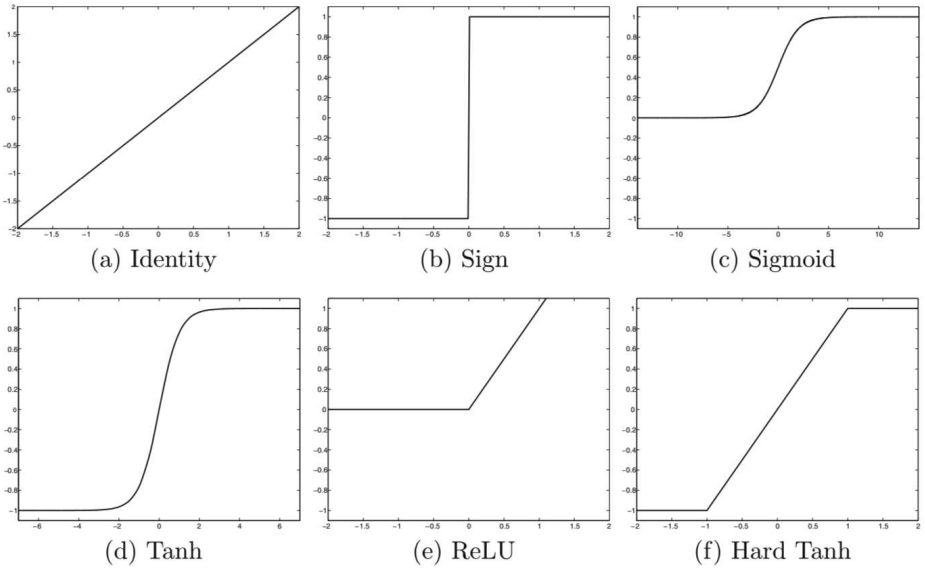


Figure 2.3: Different activation functions, [1]

While the linear activation function in a) is often used in the output node for prediction real valued targets, non-linear activation functions are very useful for predicting probabilities or in multilayer networks, as discussed in Section 2.1.2.
Multilayer networks allow to combine activation functions and use more advanced activation functions, such as for example the softmax function. The i-th output is defined as

$$\phi(\boldsymbol{v})_i = \frac{e^{v_i}}{\sum_{j=1}^{N} e^{v_j}}, \quad \forall i \in \{1, 2, \ldots, k\}.$$

In a multiclass classification problem with k different classes the final hidden layer inputs data in the sofmax layer. This final layer converts them to outputs, corresponded to the probabilities of the k classes. An example of a softmax activation function with three classes is illustrated in Figure 2.4.

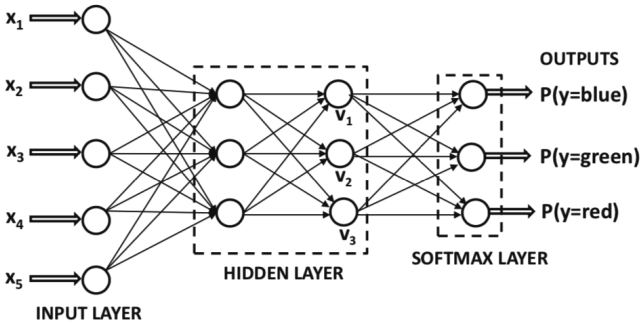
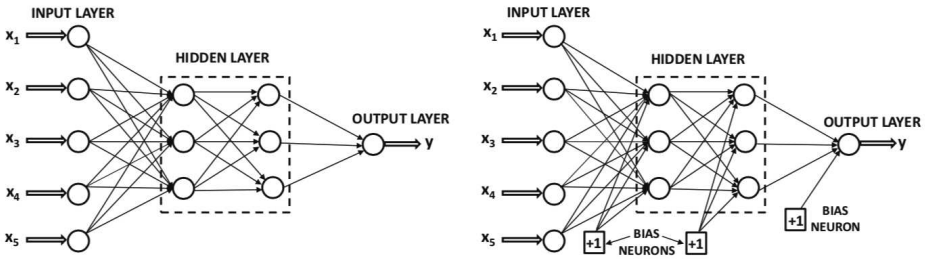


Figure 2.4: Example of classification net using the softmax activation function, [1]

2.1.2. Multilayer Neural Networks

In contrast to the perceptron, multilayer neural networks contain additional computational layers between the input and output layer, referred to as hidden layers. According to the default architecture of a feedforward neural network all nodes in each layer are connected to those in the next layer. Partially connected networks, where not every node is connected to every node in the next layer, are reffered as to sparse neural networks. Additional bias neurons can not only be used in the output layer but also in multiple hidden layers. Basic architectures of multilayer networks containing two hidden layers with and without bias are shown in Figure 2.5. Once again the input layer is not counted, because no computations are performed there.



(a) No bias neurons (b) With bias neurons

Figure 2.5: Example of a multilayer network with two hidden layers, [1]

Formally we can define a multilayer neural network as follows:

Definition 2.2 (Multilayer Neural Network)**.** Let $L, N_0, N_1, \ldots, N_L \in \mathbb{N}, \phi : \mathbb{R} \to \mathbb{R}$ and for any $l = 1, 2, \ldots, L$ let $A_l : \mathbb{R}^{N_{l-1}} \to \mathbb{R}^{N_l}$ be an affine function. Then a function

$F : \mathbb{R}^{N_0} \to \mathbb{R}^{N_L}$ defined as

$$F(x) = A_l \circ F_{L-1} \circ \cdots \circ F_1 \text{ with } F_l = \phi \circ A_l \text{ for } l = 1, 2, \ldots, L-1$$

is called a (feedforward) neural network. Hence the activation function ϕ is applied componentwise. L denotes the number of layers, $N_1, N_2, \ldots, N_{L-1}$ denote the dimension of the hidden layers and N_0, N_L of the input and output layers, respectively. For any $l = 1, 2, \ldots, L$ the affine function A_l is given as $A_l(x) = W^l x + b^l$ for some $W^l \in \mathbb{R}^{N_l \times N_{l-1}}$ and $b^l \in \mathbb{R}^{N_l}$. For any $i = 1, 2, \ldots, N_l,\ j = 1, 2, \ldots, N_{l-1}$ the number $W^l_{i,j}$ is interpreted as the weight of the edge connecting the node i of layer $l-1$ to node j of layer l. The number of non-zero weights of a network is the sum of the number of non-zero entries of the matrices $W^l,\ l = 1, 2, \ldots, L$ and vectors $b^l,\ l = 1, 2, \ldots, L$. [3]

The question of depth and width of a network has always been of highest priority for its performance. In 1989 [17] showed that for every non-constant and bounded activation function even standard neural networks with only a single hidden layer can approximate any function on any compact set arbitrarily well. Later in 1991 Hornik showed in [16]:

Theorem 2.1 (Universal Approximation)**.** *Suppose ϕ is non-constant and bounded. Then the following statement holds:*

- *For any finite measure μ on $(\mathbb{R}^{N_0}, \mathcal{B}(\mathbb{R}^{N_0}))$ and $1 \leq p < \infty$, the set of neural networks mapping from $\mathbb{R}^{N_0} \to \mathbb{R}^{N_1}$ is dense in $L^p(\mathbb{R}^{N_0}, \mu)$.*
- *If in addition $\phi \in C(\mathbb{R})$, then the set of neural networks mapping from $\mathbb{R}^{N_0} \to \mathbb{R}^{N_1}$ is dense in $C(\mathbb{R}^{N_0})$ for the topology of uniform convergence on compact sets.*

Even though this theorem states that we can represent every function it does not necessarily mean that we can learn it. Thus, the architecture plays an important role when it comes to performance. As we will see in more detail later on, we often prefer deep networks to wide ones, since they are often faster to train.

2.2. Learning Neural Networks

In the previous section we discussed the basic architectures of neural networks and how they map an input $\boldsymbol{x}$ to an output $\hat{y} = \hat{f}(\boldsymbol{x}, \boldsymbol{\theta})$ in order to approximate a unknown function $f(\cdot)$ that represents the targeted variable $y = f(\boldsymbol{x})$. Finding the best approximation is achieved by optimizing the parameters $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_N)$ which is referred to as training the neural network. In this section we discuss how to find the best weights for the approximation.

2.2.1. Maximum Likelihood Estimation

In order to predict y_i given $\boldsymbol{x}_i,\ i = 1, 2, \ldots, N$, neural networks approximate a function $f(\cdot)$, by estimating a conditional probabilities $p(y_i|\boldsymbol{x}_i; \boldsymbol{\theta}),\ i = 1, 2, \ldots, N$. Hence the maximum likelihood estimator $\boldsymbol{\theta}_{ML}$ of $\boldsymbol{\theta}$ yields the best approximation for the true probability [12].

Definition 2.3 (Maximum Likelihood Estimator)**.** Let $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N)$ be identically drawn from a generating probability distribution $p(\boldsymbol{x})$ and $p(\boldsymbol{x}; \boldsymbol{\theta})$ ba a parametric family of probability distribution over the same space indexed by $\boldsymbol{\theta}$. Then the maximum likelihood estimator for $\boldsymbol{\theta}$ is given by

$$\begin{aligned}\boldsymbol{\theta}_{ML} &= \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{x}; \boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{N} p(\boldsymbol{x}_i, \boldsymbol{\theta})\end{aligned}$$

Taking the logarithm of the likelihood does not change the argmax, thus yields the equivalent but more convenient optimization problem:

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \ln(p(\boldsymbol{x}_i; \boldsymbol{\theta}))$$

To obtain a version that is expressed as an expectation with respect to the empirical distribution $\hat{p}$ we divide the right hand side by N and get:

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\hat{p}} \left[\ln(p(\boldsymbol{x}; \boldsymbol{\theta}))\right].$$

The maximum likelihood estimator can be interpreted as finding the best approximation of the true probability distribution $p(\boldsymbol{x})$, hence we can equivalently minimize the KL divergence.

Definition 2.4 (Kullbeck–Leibler Divergence)**.** The Kullbeck–Leibler divergence between two probability distributions $\hat{p}(\boldsymbol{x})$ and $p(\boldsymbol{x}; \boldsymbol{\theta})$ is given by:

$$D_{KL}(\hat{p}(\boldsymbol{x}) \mid\mid p(\boldsymbol{x}; \boldsymbol{\theta})) = \mathbb{E}_{\hat{p}} \left[\ln(\hat{p}(\boldsymbol{x})) - \ln(p(\boldsymbol{x}; \boldsymbol{\theta}))\right].$$

Since $\ln(\hat{p}(\boldsymbol{x}))$ is independent of $\boldsymbol{\theta}$, minimizing the KL divergence is equivalent to only minimizing

$$-\mathbb{E}_{\hat{p}} \left[\ln(p(\boldsymbol{x}; \boldsymbol{\theta}))\right],$$

which is referred to as cross-entropy between the empirical distribution $\hat{p}(\boldsymbol{x})$ defined by the training set and the distribution $p(\boldsymbol{x}; \boldsymbol{\theta})$ defined by the model.
Even though for some easy problems, such as linear regression, we can simple calculate the optimal parameters $\boldsymbol{\theta}$, in practice there is almost never an analytical solution for that. Hence, we must use the log-likelihood function $\sum_{i=1}^{N} \ln(p(\boldsymbol{x}_i; \boldsymbol{\theta}))$ to find the optimum.

2.2.2. Loss Functions

In order to optimize the weights $\boldsymbol{w}$ we first define a loss function $L(\boldsymbol{x}_i, y_i, \boldsymbol{w})$ that measures the quality of each prediction $\hat{y}_i$ for the corresponding feature vector $\boldsymbol{x}_i$, $i = 1, 2, \ldots, N$. Due to the fact that minimizing the cross-entropy or negative log-likelihood yields the maximum likelihood estimator, it is the most used loss function.
The exact loss function depends on the problem and therefore on the type of output node.

Linear Regression If we see the linear regression algorithm not as a mapping from $\boldsymbol{x}$ to one single prediction $\hat{y}$ but as an algorithm producing a conditional distribution $p(y|\boldsymbol{x})$ it can be shown, that maximizing the log-likelihood with respect to the weights $\boldsymbol{w}$ yields the same estimator as minimizing the mean squared error,

$$MSE = \frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|^2.$$

Even though the two criteria have different minima, the location of the optimum remains the same.

Classification As already discussed earlier, for a K-class classifier we use the softmax activation function given by

$$\hat{p}_j = p(y = j|\boldsymbol{x};\boldsymbol{\theta}) = \text{softmax}(x_j) = \frac{e^{x_j}}{\sum_{i=1}^{K} e^{x_i}}$$

with K nodes estimating the K class probabilities $p(y = j|\boldsymbol{x}) = \hat{p}_j$ for $j = 1, 2, \ldots, K$. For this reason the target probabilities can be written as

$$p(y|\boldsymbol{x};\boldsymbol{\theta}) = \prod_{j=1}^{K} \hat{p}_j^{\boldsymbol{y}_j}$$

with $\boldsymbol{y}_j$ being a vector with all elements equal to zero except the j-th one which equals to one. Given these representations we can determine the corresponding loss function as follows:

$$L(\boldsymbol{y}, \boldsymbol{x};\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{K} y_{i,j}\ln(p(y_{i,j}|\boldsymbol{x}_i;\boldsymbol{\theta})) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{K} y_{i,j}\ln\hat{p}_{i,j}$$

For binary classification we only need to estimate one class probability $p(y = 1|\boldsymbol{x};\boldsymbol{\theta}) = \hat{p}$, which simplifies the loss function to :

$$\begin{aligned} L(\boldsymbol{y}, \boldsymbol{x};\boldsymbol{\theta}) &= -\frac{1}{n}\sum_{i=1}^{n} y_i\ln(p(y_{i,j}|\boldsymbol{x}_i;\boldsymbol{\theta})) + (1 - y_i)\ln(1 - p(y_{i,j}|\boldsymbol{x}_i;\boldsymbol{\theta})) \\ &= -\frac{1}{n}\sum_{i=1}^{n} y_i\ln(\hat{p}_i) + (1 - y_i)\ln(1 - \hat{p}_i) \end{aligned}$$

2.3. Optimization for Neural Networks

2.3.1. Backpropagation

In the last sections we have discussed how we can estimate a target y by a neural network output $\hat{y} = \hat{f}(\boldsymbol{x};\boldsymbol{\theta})$. Further we introduced a way to measure the quality of these predictions, according to which we now want to optimize the parameters $\boldsymbol{\theta}$. In order

to understand how changing the weights at different levels affects the loss $L(\boldsymbol{y}, \boldsymbol{x}; \boldsymbol{\theta})$, we need the gradients $\nabla L(\boldsymbol{y}, \boldsymbol{x}; \boldsymbol{\theta})$. These gradients are calculated by the backpropagation algorithm [33], which leverages the chain rule from differential calculus and contains two main phases:

- **Forward phase:** In this phase the input $\boldsymbol{x}$ is fed into the network, propagated through the hidden layers to produce an output $\hat{y}$, by using the current set of weights. The output can be compared to the actual target which yields the loss $L(\boldsymbol{y}, \boldsymbol{x}; \boldsymbol{\theta})$. In order to update the weights we then propagate the information from the loss backward in the network, which is done in the second phase by calculating the gradients $\nabla L(\boldsymbol{y}, \boldsymbol{x}; \boldsymbol{\theta})$.
- **Backward phase:** In this phase, starting from the output node, the gradients $\nabla L(\boldsymbol{y}, \boldsymbol{x}; \boldsymbol{\theta})$ with respect to the different weights are calculated, using the chain rule. Afterwards the weights are updated and the whole process starts from the beginning.

Even though backpropagation is often misunderstood as the whole learning algorithm for neural networks, it only refers to computing the gradients.
In order to understand backpropagation, let $h_1, h_2, \ldots, h_k$ be a sequence of hidden units, followed by an output o, that was used to compute the loss L. Furthermore, let $w_{(r,r-1)}$ be the weights between to hidden units h_r and h_{r-1}. In case only one path exists between h_1 to o the gradient of the loss with respect to any of these weights can be easily derived using the chain rule as follows, [1]:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \frac{\partial L}{\partial o} \cdot \left[\frac{\partial o}{\partial h_k} \prod_{i=1}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right] \frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} \qquad \forall r \in \{1, 2, \ldots, k\}.$$

As seen in Figure 2.5 in general there are multiple possible paths from h_1 to o. Hence, we have to generalize the above expression to the case where a set $\mathcal{P}$ contains all paths from h_r to o. Using a multivariable chain rule yields

$$\begin{aligned} \frac{\partial L}{\partial w_{(h_{r-1},h_r)}} &= \frac{\partial L}{\partial o} \cdot \left[\sum_{\{h_r, h_{r+1}, \ldots, h_k, o\} \in \mathcal{P}} \frac{\partial o}{\partial h_k} \prod_{i=1}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right] \frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} \\ &= \frac{\partial L}{\partial h_r} \frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} \qquad \forall r \in \{1, 2, \ldots, k\}. \end{aligned}$$

While $\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}}$ on the right hand side can be computed straightforward and will be discussed later, $\frac{\partial L}{\partial h_r}$ gets calculated recursively, because of the potentially high number of paths.
Beginning with nodes h_k closest to the output o and recursively going backward to earlier nodes the multivariable chain rule yields

$$\frac{\partial L}{\partial h_r} = \sum_{\{h | after\ h_r\}} \frac{\partial L}{\partial h} \frac{\partial h}{\partial h_r} \tag{2.1}$$

Since each h is in a later layer then h_r, $\frac{\partial L}{\partial h}$ has already been computed during a previous recursion step. However, we still need to compute $\frac{\partial h}{\partial h_r}$. Thereto let $w_{(h_r,h)}$ be the weights between h_r and h and let a_h be the value computed in hidden unit h just before applying the activation function ϕ, i.e. $h = \phi(a_h)$. Using the chain rule, $\frac{\partial h}{\partial h_r}$ can be derived as

$$\frac{\partial h}{\partial h_r} = \frac{\partial h}{\partial a_h}\frac{\partial a_h}{\partial h_r} = \frac{\partial \phi(a_h)}{\partial a_h} w_{(h_r,h)} = \phi'(a_h) w_{(h_r,h)}. \tag{2.2}$$

Plugging (2.1) into (2.2) yields

$$\frac{\partial L}{\partial h_r} = \sum_{\{h:after\ h_r\}} \frac{\partial L}{\partial h} \phi'(a_h) w_{(h_r,h)}.$$

Finally, $\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}}$ can be computed as

$$\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} = \frac{\partial h_r}{\partial a_{h_r}} \frac{\partial a_{h_r}}{\partial w_{(h_{r-1},h_r)}} = \phi'(a_{h_r}) h_{r-1}$$

Using this recursion gradients are computed in a backward direction and every node is processed exactly one time. As one of many alternative ways to compute the gradients we could also use the pre-activation values a_h instead of the nodes h as "chain" variables. However, these different computation ways are equivalent in terms on the final result [1].

2.3.2. Gradient Descent Algorithms

To train a model we have to constantly update the weights to find their optimum, but so far, we only described how to compute the partial derivatives of the loss function through the network. In this section we focus on different approaches to use these gradients for optimizing the weights.

The most basic algorithm known is referred to as gradient descent. We can imagine the quality of our predictions measured by the loss function as a landscape similar to a golf course. The hills represent locations (weights) with high prediction errors,whereas valleys represent locations with small errors, respectively. The goal is to move the weights as quickly as possible to areas of low errors. We therefore update the weights in small steps in the direction of the negative gradient of the loss function with respect to the weights:

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta}_{old} - \epsilon \nabla L(\boldsymbol{y}, \boldsymbol{x}; \boldsymbol{\theta}_{old}),$$

where ϵ is referred to as the learning rate. However gradient descent is hard to use in practice, since in non-linear problems it often gets stuck at a local minima. Furthermore, it can be very slow on big training sets, because it computes the gradient on the full batch.

That is why nearly all deep learning algorithms are using a stochastic gradient decent (SGD) approach. In order to speed the process up SGD computes the gradients on

small samples $\mathcal{B} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$ of the training set, called minibatches. The size m of these minibatches is referred to as batch size and normally ranges from one to a few hundred observations. It is possible to obtain an unbiased estimate of the gradient by taking the average gradient of each example on a minibatch [12, Chapter 8]. The weights are then updated every time after the average gradient of each example was computed on the minibatch as follows:

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta}_{old} - \epsilon \frac{1}{m} \nabla \sum_{i=1}^{m} L(\boldsymbol{y}, \boldsymbol{x}; \boldsymbol{\theta}_{old})$$

Ever though in general SGD already yields reasonable results there are still some improvements in terms of speed using non-constant learning rates.

Momentum On the one hand we can improve the learning process by using momentum. We introduce an additional variable called velocity that specifies in which direction and with which speed the parameters move through the parameter space. Formally, the update rule is given by:

$$\begin{aligned} \boldsymbol{\nu}_{new} &= \alpha \boldsymbol{\nu}_{old} - \epsilon \frac{1}{m} \nabla \sum_{i=1}^{m} L(\boldsymbol{y}, \boldsymbol{x}; \boldsymbol{\theta}) \\ \boldsymbol{\theta}_{new} &= \boldsymbol{\theta}_{old} + \boldsymbol{\nu}_{new}. \end{aligned}$$

The larger α is relative to ϵ, the more previous gradients affect the current direction. More detailed information how to use momentum for improving SGD can be found in [12, Section 8.3.2].

Adaptive Learning Rate On the other hand we can use a non-constant learning rate to speed up the learning process. The purpose of an adaptive learning rate is to jump far when the gradient is high and to jump in small steps when the gradient is small. The reason behind this is that gradients are high when the weights are far away from the optimum and small when the weights are already near to the optimum. This technique not only helps to find the optimal area for the weights faster, but also prevent them from jumping out of this area again.
We now introduce some of the most important algorithms that use the mini-batch approach together with an adaptive learning rate:

- The Adagrad algorithm uses a different learning rate for every parameter. To be more precise, the general learning rate η gets modified for each parameter θ_i based on the past gradients that have been computed for θ_i. This yields

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot \nabla_{\theta_{t,i}} L(\theta_{t,i}),$$

  where $G_t \in \mathbb{R}^{d \times d}$ is a diagonal matrix containing the sum of squares of the gradients computed for $\theta_{t,i}$ up to time t as diagonal elements and ϵ is a smoothing term to

avoid division by zero. Typical values for ϵ are around 10^{-8}. Since the squared gradients are non-negative the sum in the denominator keeps growing during the training process. Hence, the learning rate can get too small to gain any additional knowledge. A solution for this problem shall be provided by the Adadelta. In order to reduce this monotonically decreasing of the learning rate only a fixed number ω of previous gradients are accumulated. Finally replacing the default learning rate η with the root mean squared error of the parameter-updates at time $t-1$,

$$RMS\left[\Delta\boldsymbol{\theta}\right]_{t-1} = \sqrt{\mathbb{E}\left[\Delta\boldsymbol{\theta}^2\right]_{t-1} + \epsilon}$$

yields the vectorized Adadelta update rule:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + -\frac{RMS\left[\Delta\boldsymbol{\theta}\right]_{t-1}}{RMS\left[\nabla_{\boldsymbol{\theta}_\bullet} L(\boldsymbol{\theta}_\bullet)\right]_t} \cdot \nabla_{\boldsymbol{\theta}_t} L(\boldsymbol{\theta}_t).$$

Keeping the default learning rate instead of $RMS\left[\Delta\boldsymbol{\theta}\right]_{t-1}$ is referred to as RMSprop algorithm.

- Nevertheless, the currently most used optimization algorithm for neural networks is the Adam (Adaptive Moment Estimation). In addition to the previous algorithms, it not only uses the exponentially decaying average of the squared past gradients $\boldsymbol{v}_t$, but also exponentially smooths the first order gradients $\boldsymbol{m}_t$ in order to incorporate momentum into the update,

$$\begin{aligned}\boldsymbol{m}_t &= \boldsymbol{\beta}_1 \boldsymbol{m}_{t-1} + (1-\boldsymbol{\beta}_1)\nabla_{\boldsymbol{\theta}_t} L(\boldsymbol{\theta}_t)\\ \boldsymbol{v}_t &= \boldsymbol{\beta}_2 \boldsymbol{v}_{t-1} + (1-\boldsymbol{\beta}_2)\nabla_{\boldsymbol{\theta}_t} L(\boldsymbol{\theta}_t)^2\end{aligned}$$

Since the estimates $\boldsymbol{m}_t$ and $\boldsymbol{v}_t$ of the first and second moment of the gradients are biased towards zero [20] we use the bias-corrected estimators:

$$\begin{aligned}\hat{\boldsymbol{m}}_t &= \frac{\boldsymbol{m}_t}{1+\boldsymbol{\beta}_1}\\ \hat{\boldsymbol{v}}_t &= \frac{\boldsymbol{v}_t}{1+\boldsymbol{\beta}_2}\end{aligned}$$

Finally, we update the parameters using these estimators and get:

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

Hence, the Adam algorithm can be seen as a combination of RMSprop and using momentum.

A more detailed review of these and even more different learning algorithms can be found in [32].

2.3.3. Parameter Initialization Strategies

Now that we know how to recursively update the weights to find an optimum we have to think about where we actually want to start, i.e. how to specify an initial point from which to begin the iterations. The initial point can determine how quickly and at what numerical cost learning converges or whether it converges at all.
Even though there is a lot of literature on initialization available there is still a lack of understanding of how different initialization approaches affect the results. Perhaps, the only thing we know with complete certainty so far is that two hidden units with the same activation function and connected to the same inputs should have different initial parameters, which is referred to as symmetry breaking. Otherwise, a deterministic learning algorithm would constantly update both of these units in the same way. Even for training algorithms capable of using stochasticity to compute different updates for different units (for example, if we use dropout layers, Section 2.4), it is usually best to initialize each unit differently [12, Section 8.4].
For situations with at most as many outputs as inputs, we could use Gram-Schmidt orthogonalization on an initial weight matrix, but in general random initialization over a high-dimensional space is computationally cheaper and unlikely to assign the same weights to any unit.
Typically, we initialize the weights to values drawn from a Gaussian or uniform distribution and the biases for each unit to constants. According to [12] the scale of the initial distribution affects the outcome of the optimization procedure and the ability of the network to generalize much more than the actual choice of Gaussian or uniform distribution. On the one hand, talking from optimization perspective, weights should be large enough to propagate information successfully, but on the other hand regularization concerns encourage making them smaller.
One possible approach to initialize the weights of a fully connected layer with m inputs and n outputs is to sample them from a Gaussian distribution with standard deviation $1/\sqrt{m}$ or from the uniform distribution $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$. In order to initialize layers with not only the same activation variance, but also the same gradient variance we can use the normalized initialization,

$$W \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right).$$

For more sophisticated initialization approaches using random orthogonal matrices or scheme called sparse initialization we refer to [12, Section 8.4] and its mentioned references.

2.3.4. Batch Normalization

Last but not least, we want to introduce an optimization strategy, referred to as Batch Normalization [18], that is not an algorithm, but rather a method of adaptive reparametrization.

During the training process of deep neural networks we update the weights of all layers simultaneously and therefore also the distribution of each layer's inputs changes. In [18] this problem, referred to as internal covariate, is addressed by normalizing layer inputs with a Batch Normalization layer. This strategy does not only allow us to use higher learning rates but also to be less careful about initialization.
The Batch Normalization transformation is defined as follows.

Definition 2.5 (Batch Normalization Layer)**.** Let $\mathcal{B} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m\}$ be a minibatch of size m. Let $\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{x}_i$ be the minibatch mean and $\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{x}_i - \mu_{\mathcal{B}})^2$ be the minibatch variance. Let $\hat{\boldsymbol{x}}_i = \frac{(\boldsymbol{x}_i - \mu_{\mathcal{B}})}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$ be the normalized values of $\boldsymbol{x}_i,\ i = 1, \ldots, m$, then the Batch Normalization layer is defined as,

$$BN_{\gamma,\beta}(\boldsymbol{x}_i) = y_i = \gamma\hat{\boldsymbol{x}}_i + \beta,$$

where γ and β have to be learned.

2.4. Generalization for Neural Networks

So far we have described how to minimize the error on the training set, which is simply an optimization problem. However, the goal of machine learning models is to perform good on new data and therefore to minimize the so called generalization error. The generalization error is defined as the expected error on new data.

2.4.1. Model Selection

Unfortunately, when training the model we do not have access to test data and can not use it to pick the best performing model. However, we can create a test set by splitting the training data into a part used for training and a part for measuring the error, referred to as validation set. Then we fit all models on the training set and pick the best one according to the evaluated error on the validation set.

Cross Validation Dividing the dataset into a fixed training set and a fixed test set can be problematic if the split either yields a very small test set or does not produce a representative test set at all. Hence, multiple procedures based on the idea of repeating the split and evaluation several times on randomly selected subsets try to solve this issue. The most common one is cross validation.
Using k-fold cross validation we split the data set into k equally sized non-overlapping subsamples $S_1, \ldots, S_k$, train the model on all but one subsample and test it on the remaining one. We repeat this procedure k times, so that every S_i for $i = 1, \ldots, k$ serve as test set exactly once. Finally, the average of all k scores is the generalization score. We select the model that has the best average score.

In order to pick the best model based on measured errors we have to think about the

bias-variance trade-off. Since we are looking at stock prices in this thesis, we are dealing with noisy data, i.e. $y_i = f(\boldsymbol{x}_i) + \epsilon_i,\ i = 1, 2, \ldots, m$ for an unknown function f and some additional observation noise ϵ_i. Usually, every ϵ_i is modeled independent of the model and satisfies $\mathbb{E}[\epsilon_i] = 0$. Then the squared error can be decomposed into the (squared) bias, variance, and noise, as follows, [1, Section 4.2]:

$$\begin{aligned}
\mathbb{E}\left[\text{MSE}\right] &= \mathbb{E}\left[\frac{1}{t}\sum_{i=i}^{t}(\hat{y}_i - y_i)^2\right] \\
&= \frac{1}{t}\sum_{i=i}^{t}\mathbb{E}\left[(\hat{y}_i - y_i)^2\right] \\
&= \frac{1}{t}\sum_{i=i}^{t}\mathbb{E}\left[((\hat{y}_i - \mathbb{E}[\hat{y}_i]) + (\mathbb{E}[\hat{y}_i] - f(\boldsymbol{x}_i)) - \epsilon_i)^2\right]. \\
&= \frac{1}{t}\sum_{i=i}^{t}\mathbb{E}\left[(\hat{y}_i - \mathbb{E}[\hat{y}_i])^2 + 2(\hat{y}_i - \mathbb{E}[\hat{y}_i])(\mathbb{E}[\hat{y}_i] - f(\boldsymbol{x}_i)) - 2(\hat{y}_i - \mathbb{E}[\hat{y}_i])\epsilon_i + \right. \\
&\qquad \left. + (\mathbb{E}[\hat{y}_i] - f(\boldsymbol{x}_i))^2 - 2(\mathbb{E}[\hat{y}_i] - f(\boldsymbol{x}_i))\epsilon_i + \epsilon_i^2\right].
\end{aligned}$$

Using that ϵ_i is centered and independent of the model as well as the fact that $(\mathbb{E}[\hat{y}_i] - f(\boldsymbol{x}_i))$ is deterministic, we get:

$$\mathbb{E}\left[\text{MSE}\right] = \underbrace{\frac{1}{t}\sum_{i=1}^{t}\mathbb{E}\left[(\hat{y}_i - \mathbb{E}\left[\hat{y}_i\right])^2\right]}_{\text{Variance}} + \underbrace{\frac{1}{t}\sum_{i=1}^{t}(\mathbb{E}[\hat{y}_i] - f(\boldsymbol{x}_i))^2}_{\text{Bias}^2} + \underbrace{\frac{\sum_{i=1}^{t}\mathbb{E}\left[\epsilon_i^2\right]}{t}}_{\text{Noise}}.$$

The variance is the key term that prevents neural networks from generalizing. Typically, the variance is higher for neural networks that have a large number of parameters. However, too few model parameters can cause a high bias because there are not sufficient degrees of freedom to model the complexities of the data distribution. Hence, stopping training too late eventually reduces bias but enlarges variance, whereas stopping training too early eventually reduces variance, but yields a higher bias.

2.4.2. Regularization Techniques

Keeping the above mentioned trade-off in mind, we present two approaches to prevent neural networks from overfitting and therefore increase their generalization capabilities.

Early Stopping One way to deal with this challenge is the early stopping method deployed by [35]. In order to find the best stopping point in time a predefined error measure on the validation set is monitored. Even though, the training set error may

still decrease during training, sometimes the test set error starts increasing again after some training steps. In that case early stopping terminates the training process after a few iterations. Due to its effectiveness in restricting the parameter space and reducing training time it is one of the most used regularization techniques in deep learning.

Dropout Another common technique to avoid overfitting is to use dropout layers in which randomly selected units are dropped. More precisely, during each training step we either keep an individual node with probability p or drop it with probability $1 - p$. Dropping a node means setting its weight to zero, which yields a smaller network with less parameters to train. As a result the training time per iteration decreases a lot, however usually almost twice as many iterations are required to converge.

3. Limit Order Book

In this chapter we first want to formulate a mathematically precise description of limit order books by giving some formal definitions and then have a look into the actual structure of LOBSTER data.

3.1. Mathematical Description

In order to define a limit order book $\mathcal{L}(t)$ we start giving some fundamental definitions, by following [13].

Definition 3.1 (Limit Order)**.** A limit order $x = (p_x, w_x, t_x)$ submitted at time t_x at price p_x and size $w_x > 0$ (respectively, $w_x < 0$) is a commitment to sell (respectively, buy) up to $|w_x|$ units of the traded asset at a price no less than (respectively, no greater than) p_x.

For a given limit order book (LOB) lot size σ and tick size π, collectively called its resolution parameters, can be defined by:

Definition 3.2 (Lot Size)**.** The lot size σ of an LOB is the smallest amount of the asset that can be traded within it. All orders must arrive with a size $w_x \in \{\pm k\sigma | k = 1, 2, \dots\}$.

Definition 3.3 (Tick Size)**.** The tick size π of an LOB is the smallest permissible price interval between different orders within it. All orders must arrive with a price that is specified to the accuracy of π.

Now we can finally define a limit order book as:

Definition 3.4 (Limit Order Book)**.** An LOB $\mathcal{L}(t)$ is the set of all active orders in a market at time t.

This set of all active orders, $\mathcal{L}(t)$, is a càdlàg process since for every new limit order x, the following holds: $x \in \mathcal{L}(t)$ and $x \notin \lim_{t' \uparrow t_x} \mathcal{L}(t')$. For further distinctions $\mathcal{L}(t)$ can be partitioned into the set of active buy orders $\mathcal{B}(t) = \{(p_x, w_x, t_x) \mid w_x < 0\}$ and the set of active sell orders $\mathcal{A}(t) = \{(p_x, w_x, t_x) \mid w_x > 0\}$. This allows us to define the depth size available for bid and ask side.

Definition 3.5 (Available Depth Size)**.** The bid-side depth available at price p and at time t is

$$n^b(p, t) := \sum_{\{x \in \mathcal{B}(t) | p_x = p\}} \omega_x$$

The ask-side depth available at price p and at time t, denoted $n^a(p, t)$, is defined similarly using $\mathcal{A}(t)$.

The depth of each level in both, bid and ask, sides is modified constantly due to limit orders, market orders and cancellations. Limit orders increase the size depth while market orders and cancellations remove liquidity from LOB and therefore reduce the depth.
Lastly, we define the terms bid price, ask price, mid price and bid-ask spread. These terms are commonly used in a variety of finance literature and get covered especially in Section 5.1.2.

Definition 3.6 (Best Bid/Ask Price)**.** The best bid price at time t is the highest stated price among active buy orders at time t,

$$b(t) \coloneqq \max_{x \in \mathcal{B}(t)} p_x,$$

whereas the best ask price at time t the lowest stated price among active sell orders at time t is,

$$a(t) \coloneqq \min_{x \in \mathcal{A}(t)} p_x.$$

Definition 3.7 (Bid-Ask Spread)**.** The difference between best ask price and best bid price at time t is denoted as bid-ask spread $s(t) \coloneqq a(t) - b(t)$.

Definition 3.8 (Midprice)**.** The midprice at time t is $m(t) \coloneqq \frac{1}{2}(a(t) + b(t))$

With these definition in mind, we now want to have a look into the actual data structure provided by NASDAQ.

3.2. Data Structure

In this section we have a closer look at the data we use and its structure. As mentioned before we are using limit order book data from the NASDAQ stock exchange. For each stock all events, such as order submissions, order cancellations and order execution as well as the state of the order book are recorded. The execution times of these events are reported in nanosecond decimal precision and the order book state does not change between two events. In general order books can be very deep and include up to 50 non-zero levels, which are levels with non-zero trading volumes, on both, ask and bid, side. However, the more levels are needed the more expensive it gets to purchase this data.
The available data contains a message and a orderbook file for each active trading day for each selected stock. The message file reports all events causing an update of the limit order book in the requested price range, while the orderbook file contains the evolution of the limit order book up to the requested number of levels. These events are categorized in seven different types and timestamped to seconds after midnight, with decimal precision of at least milliseconds and up to nanoseconds depending on the requested period. In Tables 3.1 and 3.2 the basic structure of message and orderbook files, as provided by LOBSTER, are described in detail.

Table 3.1: Structure of a message file

Time (sec)	Event Type	Order ID	Size	Price	Direction
⋮	⋮	⋮	⋮	⋮	⋮
34713.685155243	1	206833312	100	118600	-1
34714.133632201	3	206833312	100	118600	-1
⋮	⋮	⋮	⋮	⋮	⋮

The columns can be explained as follows:

- Time: Seconds after midnight with decimal precision of at least milliseconds and up to nanoseconds depending on the period requested
- Event Type:
  1. Submission of a new limit order
  2. Cancellation (partial deletion of a limit order)
  3. Deletion (total deletion of a limit order)
  4. Execution of a visible limit order
  5. Execution of a hidden limit order
  6. Indicates a cross trade, e.g. auction trade
  7. Trading halt indicator (detailed information below)
- Order ID: Unique order reference number
- Size: Number of shares
- Price: Dollar price times 10000 (i.e. a stock price of $91.14 is given by 911400)
- Direction:
  - -1: Sell limit order
  - 1: Buy limit order
  - Note: Execution of a sell (buy) limit order corresponds to a buyer (seller) initiated trade, i.e. buy (sell) trade.

The numbers in the head of the orderbook file represent the different levels.

Table 3.2: Structure of an orderbook file

Ask Prize 1	**Ask Size 1**	**Bid Prize 1**	**Bid Size 1**	**Ask Prize 1**	**Ask Size 1**	**Bid Prize 1**	**Bid Size 1**	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1186600	9484	118500	8800	118700	22700	118400	14930	...
1186600	9484	118500	8800	118700	22700	118400	14930	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Further information regarding the data structure can be found on [24].

4. Volatility Estimation using Noisy High-Frequency Data

Volatility is a fundamental factor in finance, especially as a parameter for any option pricing model, such as the famous Black-Scholes model. In the literature, authors often distinguish between historical or realized and implied volatility. In contrast to historical volatility, which looks at actual asset prices, implied volatility looks ahead.

- **Realized volatility** refers to the volatility experienced by a financial instrument over a certain period of time.
- **Implied volatility** is the market's forecast of future fluctuations of a financial instrument. It can be determined by using an option pricing model, as it is the only factor that can not be directly observed on the markets.

Since volatility can not be directly observed on the markets it is a common practice in finance to estimate volatility from the sum of frequently sampled squared returns. However, this estimator has some weaknesses in presence of market microstructure. Hence, we present an estimation approach that takes advantage of the rich sources in tick-by-tick data while preserving the continuous-time assumption on the underlying returns, as proposed by [36].

4.1. Quadratic Variation and Realized Volatility

In this section, we discuss the estimation of volatility using high frequency data, such as LOB data. We introduce a traditional approach to estimate realized volatility as well as its weaknesses in presence of market microstructure. Then, we present the two scales realized volatility estimator (TSRV), as one of several approaches, especially designed for capturing noise in high frequency data.
Let S_t denote the price process of a security over an interval $[0, T]$, and suppose that the process $X_t = \log(S_t)$ follows an Itô process,

$$dX_t = \mu_t dt + \sigma_t dB_t, \tag{4.1}$$

where B_t is standard Brownian motion. The drift μ_t and the instantaneous variance σ_t^2 are continuous stochastic processes.
Then parameter of interest is the integrated variance

$$\langle X \rangle_t = \int_0^T \sigma_t^2 dt,$$

where $\langle X \rangle_t$ denotes the quadratic variation. According to well known theoretical results about stochastic processes, the quadratic variation of any Itô process can be expressed as a limit in probability,

$$\sum_{i=0}^{n-1} (X_{t_{i+1}} - X_{t_i})^2 \xrightarrow{p} \int_0^T \sigma_t^2 dt,$$

where $0 = t_o < t_1 < \cdots < t_{n-1} < t_n = T$. Hence, a natural approach to estimate the integrated variance is to use the sum of squared returns,

$$[X, X]_T \triangleq \sum_{i=0}^{n-1} \left(X_{t_{i+1}} - X_{t_i}\right)^2.$$

This estimator is called realized variance and in theory the error of the realized variance should diminish as the sampling frequency increases.
However, it has been found empirically that this estimator is not robust when the sampling interval is small. In fact, the estimator is highly biased in presence of market microstructure, including, but not limited to, the existence of the bid–ask spread [36]. Therefore, many authors sample over longer time horizons to obtain more reasonable estimates. Although this approach is widely used in the literature, there comes a price with it, namely discarding a lot of data. Since it is hard to accept that sometimes discarding almost 99% of the recorded data would be necessary [14], we present the two scales realized volatility estimator proposed in [36]. This estimator not only models the market microstructure as observation error, but also uses two different sampling scales over all data.

4.2. Realized Volatility under Market Microstructure Noise

4.2.1. Setup and Sparse Sampling

Let Y_t be the log-return process as actually observed at sampling times $0 = t_o < t_1 < \cdots < t_{n-1} < t_n = T$, such that

$$Y_{t_i} = X_{t_i} + \epsilon_{t_i}, \quad \forall i \in \{0, 1, \ldots, n\} \tag{4.2}$$

with X_{t_i} following (4.1). The noise ϵ_{t_i} satisfies

$$\mathbb{E}[\epsilon_{t_i}] = 0 \text{ and } Var(\epsilon_{t_i}) = \mathbb{E}[\epsilon^2],$$

with $\epsilon \perp\!\!\!\perp X$, where $\perp\!\!\!\perp$ denotes independence between two random quantities.

Definition 4.1 (Full Grid)**.** The full grid $\mathcal{G}$ containing all observations points is given by,

$$\mathcal{G} \coloneqq \{t_0, t_1, \ldots, t_n\}.$$

Definition 4.2 (Grid)**.** For an arbitrary grid $\mathcal{H} \subseteq \mathcal{G}$, denoting successive elements of the full grid $\mathcal{G}$ and $t_i \in \mathcal{H}$ we denote the proceeding and following elements in $\mathcal{H}$ by t_{i-} and t_{i+}, respectively. We always denote the i^{th} point in $\mathcal{G}$ with t_i, thus for $\mathcal{H} = \mathcal{G}$ we have $t_{i-1} = t_{i-}$ and $t_{i+1} = t_{i+}$. Finally, the number of time increments $(t_i, t_{i+1}]$, such that both endpoints are contained in $\mathcal{H}$ is given by,

$$|\mathcal{H}| = (\#\text{ points in grid } \mathcal{H}) - 1.$$

Hence, we can conclude that the size of $\mathcal{G}$ is given by, $|\mathcal{G}| = n$.

Definition 4.3 (Observed Quadratic Variation)**.** For a generic process Z, such as X or Y, and an arbitrary grid $\mathcal{H}$ the observed quadratic variation $[Z]_t^{\mathcal{H}}$ is defined as,

$$[Z]_t^{\mathcal{H}} \coloneqq \sum_{t_j, t_{j+} \in \mathcal{H}, t_{j+} \leq t} (Z_{t_{j+}} - Z_{t_j})^2.$$

On the full grid $\mathcal{G}$ the quadratic variation is defined as,

$$[Z]_t^{(all)} = [Z]_t^{\mathcal{G}} \coloneqq \sum_{t_j, t_{j+1} \in \mathcal{H}, t_{j+1} \leq t} (Z_{t_{j+1}} - Z_{t_j})^2 = \sum_{j=0}^{n-1} (\Delta Z_{t_j})^2,$$

where $\Delta Z_{t_j} \coloneqq (Z_{t_{j+1}} - Z_{t_j})$.

The observed quadratic covariations are defined similarly.

Definition 4.4 (Observed Quadratic Covariations)**.** The observed quadratic corvariation for two processes X and Y on an arbitrary grid $\mathcal{H} \subseteq \mathcal{G}$ is defined as,

$$[X, Y]_t^{\mathcal{H}} \coloneqq \sum_{t_j, t_{j+} \in \mathcal{H}, t_{j+} \leq t} (X_{t_{j+}} - X_{t_j})^2 (Y_{t_{j+}} - Y_{t_j})^2.$$

On the full grid $\mathcal{G}$ it is defined as,

$$[X, Y]_t^{\mathcal{G}} \coloneqq \sum_{j=0}^{n-1} (\Delta X_{t_j})^2 (\Delta Y_{t_j})^2.$$

Finally, in our asymptotic assumption we assume that as the number of observations in $[0, T]$ tends to infinity, the maximum distance in time between two consecutive observations tends to 0,

$$\max_i \Delta t_i \to 0 \text{ as } n \to \infty.$$

4.2.2. Biased Realized Volatility

Given the additive model $Y_{t_i} = X_{t_i} + \epsilon_{t_i},\ i = 0, 1, \ldots, n$ the realized variance based on the observed log-returns Y_{t_i} can be computed as follows.

$$\begin{aligned}
[Y]_T^{\mathcal{G}} &= \sum_{i=0}^{n-1} (Y_{t_{i+1}} - Y_{t_i})^2 \\
&= \sum_{i=0}^{n-1} (X_{t_{i+1}} + \epsilon_{t_{i+1}} - X_{t_i} - \epsilon_{t_i})^2 \\
&= \sum_{i=0}^{n-1} \left((X_{t_{i+1}} - X_{t_i})^2 + 2(X_{t_{i+1}} - X_{t_i})(\epsilon_{t_{i+1}} - \epsilon_{t_i}) + (\epsilon_{t_{i+1}} - \epsilon_{t_i})^2 \right) \\
&= [X]_T^{\mathcal{G}} + 2[X, \epsilon]_T^{\mathcal{G}} + [\epsilon]_T^{\mathcal{G}}
\end{aligned}$$

Proposition 4.1. *Assuming* $\mathbb{E}[\epsilon^4] < \infty$*, the conditional expectation and conditional variance of* $[Y]_T^{\mathcal{G}}$ *are given by,*

$$\mathbb{E}\left[[Y]_T^{\mathcal{G}}|X\right] = [X]_T^{\mathcal{G}} + 2n\mathbb{E}[\epsilon^2] \tag{4.3}$$

$$Var\left([Y]_T^{\mathcal{G}}|X\right) = 4n\mathbb{E}[\epsilon^4] + \mathcal{O}_p(1). \tag{4.4}$$

Proof. Using the result above and $\epsilon \perp\!\!\!\perp X$, we can compute the conditional expectation of $[Y]_T^{\mathcal{G}}$ as follows,

$$\begin{aligned}\mathbb{E}\left[[Y]_T^{\mathcal{G}}|X\right] &= \mathbb{E}\left[[X]_T^{\mathcal{G}}|X\right] + 2\mathbb{E}\left[[X,\epsilon]_T^{\mathcal{G}}|X\right] + \mathbb{E}\left[[\epsilon]_T^{\mathcal{G}}|X\right] \\ &= [X]_T^{\mathcal{G}} + 2\mathbb{E}\left[[X,\epsilon]_T^{\mathcal{G}}|X\right] + \mathbb{E}\left[[\epsilon]_T^{\mathcal{G}}\right]\end{aligned}$$

Using properties of the conditional expectations as well as $\mathbb{E}[\epsilon_{t_i}] = 0,\ \forall t_i \in [0,T]$ and $\mathbb{E}[\epsilon_{t_i}\epsilon_{t_j}] = 0$ for $i \neq j$ yields following representations,

$$\mathbb{E}\left[[X,\epsilon]_T^{\mathcal{G}}|X\right] = \sum_{i=0}^{n-1}(X_{t_{i+1}} - X_{t_i})^2\mathbb{E}\left[(\epsilon_{t_{i+1}} - \epsilon_{t_i})^2\right] = 0$$

$$\mathbb{E}\left[[\epsilon]_T^{\mathcal{G}}\right] = \sum_{i=0}^{n-1}\mathbb{E}[\epsilon_{t_{i+1}}^2] + \mathbb{E}[\epsilon_{t_{i+1}}\epsilon_{t_i}] + \mathbb{E}[\epsilon_{t_i}^2] = 2n\mathbb{E}[\epsilon^2]$$

Hence, the conditional expectation is given by,

$$\mathbb{E}\left[[Y]_T^{\mathcal{G}}X\right] = [X]_T^{\mathcal{G}} + 2n\mathbb{E}[\epsilon^2].$$

For the proof regarding the conditional variance we refer to [36, Appendix A.1] and [15]. □

Equations (4.3) and (4.4) show that in presence of microstructure the realized variance $[Y]_T^{\mathcal{G}}$ is not a reliable estimator for the true quadratic variation $[X]_T^{\mathcal{G}}$. In fact, as n becomes large both its bias and its variance increase. Nevertheless, Equation (4.3) provides a consistent estimator for the variance of the noise,

$$\widehat{\mathbb{E}\left[\epsilon^2\right]} = \frac{1}{2n}[Y]_T^{\mathcal{G}},$$

that follows the asymptotic distribution,

$$n^{1/2}\left(\widehat{\mathbb{E}\left[\epsilon^2\right]} - \mathbb{E}[\epsilon^2]\right) \xrightarrow{\mathcal{L}} N\left(0, \mathbb{E}[\epsilon^2]\right), \qquad \text{as } n \to \infty.$$

According to [36], a consistent estimator of the asymptotic variance of $\mathbb{E}[\epsilon^2]$ is then given by,

$$\widehat{\mathbb{E}[\epsilon^4]} = \frac{1}{2n}\sum_{i=0}^{n-1}(\Delta Y_{t_i})^4 - 3(\widehat{\mathbb{E}[\epsilon^2]})^2.$$

4.2.3. Two Scales Realized Volatility Estimator (TSRV)

In the previous section we showed that increasing the amount of used data would increase the bias, thus we could benefit from infrequently sampled data. And yet every basic statistic lecture tells us not to do that. As a possible solution we present the two scales realized volatility estimator.
The two scales realized volatility estimator samples over two time horizons by selecting a number of subgrids of the original grid of observation times and afterwards averages the estimators derived from the subgrids.
Formally, we partion the full grid $\mathcal{G}$ into K non-overlapping subgrids $\mathcal{G}^{(k)}$, $k = 1, \dots, K$ satisfying,

$$\mathcal{G} = \bigcup_{k=1}^{K} \mathcal{G}^{(k)},$$

where $\mathcal{G}^{(k)} \cap \mathcal{G}^{(l)} = \emptyset$ for $k \neq l$. An intuitive approach of selecting these K subgrids is to start with t_{k-1} and then pick every K^{th} sample point until T. This yields,

$$\mathcal{G}^{(k)} = \{t_{k-1}, t_{k-1+K}, t_{k-1+2*K}, \dots, t_{k-1+n_k*K}\},$$

for $k = 1, 2, \dots, K$ and an integer n_k making t_{k-1+n_k*K} the last element in $\mathcal{G}^{(k)}$. Using these subgrids we can define an averaging estimator.

Definition 4.5 (Averaging Estimator)**.** The averaging estimator is given by,

$$[Y]_T^{(avg)} := \frac{1}{K} \sum_{k=1}^{K} [Y]_T^{(k)},$$

where $[Y]_T^{(k)} := \sum_{t_j, t_{j+} \in \mathcal{G}^{(k)}} (Y_{t_{j+}} - Y_{t_j})^2$ is the observed realized variance on the k^{th} subgrid $\mathcal{G}^{(k)}$.

Using this estimator we have to deal with two sources of error.

- **Error due to the noise $[\boldsymbol{Y}]_{\boldsymbol{T}}^{(\boldsymbol{avg})} - [\boldsymbol{X}]_{\boldsymbol{T}}^{(\boldsymbol{avg})}$**
  Recalling equations (4.3) and (4.4) we have,

$$\mathbb{E}\left[[Y]_T^{(avg)} | X\right] = [X]_T^{(avg)} + 2\bar{n}\mathbb{E}\left[\epsilon^2\right]$$
$$Var\left([Y]_T^{(avg)} | X\right) = 4\frac{\bar{n}}{K}\mathbb{E}\left[\epsilon^4\right] + \mathcal{O}_p(\frac{1}{K}),$$

  where $\bar{n} = \frac{1}{K}\sum_{k=1}^{K} n_k = \frac{n-K+1}{K}$. These expressions allow us to deduce the conditional asymptotic distribution for the estimator $[Y]_T^{(avg)}$.

  Theorem 4.1. *Suppose X follows (4.1) and Y is related to X through (4.2). Additionally, we suppose that $\mathbb{E}[\epsilon^4] < \infty$ holds and that t_i and t_{i+1} are not in the same subgrid for any $i \in \{0, 1, \dots, n\}$. Then conditioned on X*

$$\sqrt{\frac{K}{\bar{n}}}\left([Y]_T^{(avg)} - [X]_T^{(avg)} - 2\bar{n}\mathbb{E}\left[\epsilon^2\right]\right) \xrightarrow{\mathcal{L}} 2\sqrt{\mathbb{E}\left[\epsilon^4\right]} Z_{\text{noise}}^{(avg)},$$

  where $Z_{\text{noise}}^{(avg)}$ is standard normally distributed, holds for $n \to \infty$.

Proof. See [36, Appendix A] □

- **Error due to the discretization effect $[X]_T^{(avg)} - \langle X \rangle_T^{(avg)}$**
  Under some additional assumption [36] the asymptotic distribution of this error is given by the following theorem.

  Theorem 4.2. *Under some assumptions the error $D_T = [X]_T^{(avg)} - \langle X \rangle_T^{(avg)}$ is asymptotically mixed normally distributed. For a random variable η and $n \to \infty$,*

  $$\sqrt{\frac{n}{K}} \left([X]_T^{(avg)} - \langle X \rangle_T^{(avg)} \right) \xrightarrow{\mathcal{L}} \eta \sqrt{T} Z_{\text{discrete}},$$

  where Z_{discrete} is normally distributed, holds.

  Proof. For all assumptions and the proof, see [36]. □

Finally, we want to combine the two error terms arising from discretization and from the observation noise. Theorem 4.1 and Theorem 4.2 yield

$$[Y]_T^{(avg)} - \langle X \rangle_T - 2\bar{n}\mathbb{E}[\epsilon^2] = \xi Z_{\text{total}} + o_p(1),$$

where Z_{total} is asymptotically standard normally distributed and independent of X and

$$\xi^2 = \underbrace{4\frac{\bar{n}}{K}\mathbb{E}[\epsilon^4]}_{\text{due to noise}} + \underbrace{T\frac{1}{\bar{n}}\eta^2}_{\text{due to discretization}}.$$

For $K = cn^{\frac{2}{3}}$ both components of ξ are equally present in the limit, whereas for other values of K one term dominates.
Even though, the average estimator already provides an improvement in asymptotic bias and asymptotic variance, it is still a biased estimator. Before we present the final bias-adjusted estimator we first investigate the optimal sampling frequency.
In order to find the optimal value $\bar{n}$ for subsampling we minimize the MSE of $[Y]_T^{(avg)}$ given by,

$$\begin{aligned} MSE &= bias^2 + \xi^2 = 4\left(\mathbb{E}[\epsilon^2]\right)^2 \bar{n}^2 + 4\frac{\bar{n}}{K}\mathbb{E}[\epsilon^4] + \frac{T}{\bar{n}}\eta^2 \\ &= 4\left(\mathbb{E}[\epsilon^2]\right)^2 \bar{n}^2 + \frac{T}{\bar{n}}\eta^2 \end{aligned}$$

Computing the first derivative w.r.t. $\bar{n}$ and setting it equal to 0 yield an optimal $\bar{n}^*$, satisfying,

$$\bar{n}^* = \left(\frac{T\eta^2}{8\left(\mathbb{E}[\epsilon^2]\right)^2} \right)^{1/3}.$$

Therefore, can use all n observations, if we use $K^* \approx \frac{n}{\bar{n}^*}$ subgrids. However for the asymptotics to hold we need $\mathbb{E}[\epsilon^2] \to 0$.

Final Estimator $\widehat{\langle X \rangle_T}$

So far we have seen that $[Y]_T^{(avg)}$ is already an improved but still biased estimator and that $\mathbb{E}[\epsilon^2]$ can be consistently approximated by,

$$\widehat{\mathbb{E}[\epsilon^2]} = \frac{1}{2n}[Y]_T^{\mathcal{G}}.$$

Hence, the bias of $[Y]_T^{(avg)}$ can be consistently estimated by $2\bar{n}\widehat{\mathbb{E}[\epsilon^2]}$. This allows us to finally define the bias adjusted two scales realized volatility estimator.

Definition 4.6 (Two Scales Realized Volatility)**.** The bias adjusted estimator for $\langle X \rangle$, called two scales realized volatility, is given by,

$$\widehat{\langle X \rangle}_T := [Y]_T^{(avg)} - \frac{\bar{n}}{n}[Y]_T^{(all)}.$$

Note, that the two time scales (avg) and (all) are combined.

Theorem 4.3. *Suppose X is an Itô process following (4.1) and Y is related to X through (4.2). Further, let $K = cn^{\frac{2}{3}}$ and suppose that some asymptotic assumptions. Then,*

$$\begin{aligned} n^{1/6}\left(\widehat{\langle X, X \rangle}_T - \langle X, X \rangle_T\right) &\xrightarrow{\mathcal{L}} N\left(0, 8c^{-2}\left(\mathbb{E}\left[\epsilon^2\right]\right)^2\right) + \eta\sqrt{T}N(0,c) \\ &= \left(8c^{-2}\left(\mathbb{E}\left[\epsilon^2\right]\right)^2 + c\eta^2 T\right)^{\frac{1}{2}} N(0,1) \end{aligned}$$

where c is a constant aiming to minimize the overall variance, holds.

Proof. For all assumptions and the proof, see [36]. □

To find the optimal sampling frequency K, we can minimize the expected asymptotic variance to obtain

$$c^* = \left(\frac{16\left(\mathbb{E}\left[\epsilon^2\right]\right)^2}{T\mathbb{E}[\eta^2]}\right)^{\frac{1}{3}}.$$

For calculation purposes estimators for $\mathbb{E}[\eta^2]$ and η are used.

5. Experiments

In this section we investigate the predictive power of neural networks trained on limit order book data by using different network architectures and comparing them, regarding performances, with commonly used algorithms. We target two of the most important tasks for investors, namely predicting price movements and forecasting volatility.

5.1. Preliminaries

Before presenting empirical results we first specify the actually available data and the features we extract out of it. Afterwards, we introduce commonly used methods to select the most important features for every experiment as well as normalization approaches.

5.1.1. Available Data

In this thesis we use data from four different stocks, in particular we use Amazon, Apple, Google and Tesla. Each data set contains the first five non-zero levels on each side of the order book. These data sets are summarized in Table 5.1.

Table 5.1: Available data

Ticker	Snapshot	Trading Days
AMZN	8.Jan.2018 - 6.Aug.2019	397
APPL	8.Jan.2018 - 8.Aug.2019	399
GOOGL	8.Jan.2018 - 23.Aug.2019	410
TSLA	8.Jan.2018 - 23.Aug.2019	410

Note, that not the whole data sets are used during the experiments presented in the following sections.

5.1.2. Handcrafted Features

In many cases the available data in its original form is not informative enough to yield reasonable results, thus we have to look for additional features to extract out of the data.
In this thesis we only use the orderbook files described in Section 3.2 and start by extracting most of the features proposed in [27]. More specifically, we use three sets of features:

- a basic set, including the prices and volumes for every level of both, bid and ask, side

- a time-insensitive set, containing the bid-ask spread, mid-price, price differences, mean prices, mean volumes and accumulated price as well as accumulated volume differences between the bid and ask orders of each level
- Finally, time-sensitive features corresponding to price and volume derivatives are extracted

A more detailed overview is represented in Table 5.2.

Table 5.2: Original feature set

Basic Set	**Description (i = level index)**
$v_1 = \{P_i^{ask},\ V_i^{ask},\ P_i^{bid},\ V_i^{bid}\}_{i=1}^{n}$	*price and volume (n levels)*

Time-insensitive Set	**Description (i = level index)**
$v_2 = \{(P_i^{ask} - P_i^{bid}),\ \frac{1}{2}(P_i^{ask} + P_i^{bid})\}_{i=1}^{n}$	*bid-ask spreads and mid-prices*
$v_3 = \{P_n^{ask} - P_1^{ask},\ P_1^{bid} - P_n^{bid},\ \lvert P_{i+1}^{ask} - P_i^{ask}\rvert,$ $\lvert P_{i+1}^{bid} - P_i^{bid}\rvert\}_{i=1}^{n-1}$	*price differences*
$v_4 = \{\frac{1}{n}\sum_{i=1}^{n} P_i^{ask},\ \frac{1}{n}\sum_{i=1}^{n} P_i^{bid},\ \frac{1}{n}\sum_{i=1}^{n} V_i^{ask},$ $\frac{1}{n}\sum_{i=1}^{n} V_i^{bid}\}$	*mean prices and volumes*
$v_5 = \{\sum_{i=1}^{n}(P_i^{ask} - P_i^{bid}),\ \sum_{i=1}^{n}(V_i^{ask} - V_i^{bid})\}$	*accumulated differences*

Time-sensitive Set	**Description (i = level index)**
$v_6 = \{dP_i^{ask}/dt,\ dV_i^{ask}/dt,\ dP_i^{bid}/dt,\ dV_i^{bid}/dt,$ $dP_1^{mid}/dt\}_{i=1}^{n}$	*price and volume derivatives*

In feature v_6 the average derivatives of price and volume are computed over the most recent 5 events as well as the changes between the current and the most recent available value. We also implement some additional features as proposed in [30] and listed in Table 5.3. These features are all time-sensitive.

Table 5.3: Additional feature set

Time-sensitive Set	**Description (i = level index)**
$v_7 = \{\frac{1}{m}\sum_{j=1}^{m} P_{i,j}^{ask},\ \frac{1}{m}\sum_{j=1}^{m} P_{i,j}^{bid},\ \frac{1}{m}\sum_{j=1}^{m} V_{i,j}^{ask},$ $\frac{1}{m}\sum_{j=1}^{m} V_{i,j}^{bid}\}_{i=1}^{n}$	*mean prices and volumes over m most recent values*
$v_8 = \{\frac{1}{m}\sum_{j=1}^{m}(P_{1,j}^{ask} - P_{1,j}^{bid}),\ \frac{1}{m}\sum_{j=1}^{m}\frac{1}{2}(P_{1,j}^{ask} + P_{1,j}^{bid})\}$	*first level mean bid-ask spread and midprice over m most recent values*
$v_9 = \{(\frac{1}{s}\sum_{j=1}^{s} P_{1,j}^{mid} - \frac{1}{l}\sum_{j=1}^{l} P_{1,j}^{mid})\}$	*trend oscillator midprice*

In feature v_7 and v_8 we compute the average prices and volumes as well as the first level bid-ask spread and midprice over the most recent 5 values. Feature v_9 is added to

capture market momentum, by computing the difference between a short term moving average over the most recent 3 values and a long term moving average over the most recent 10 values. Additional technical features to add can be found in [30]

5.1.3. Feature Selection Methods

In Machine Learning projects we often deal with huge data sets containing a tremendous amount of different variables. Some of them may be redundant or even decrease the accuracy of models. Hence, selecting meaningful features and consequently reducing the input's dimensionality is an important step in the extraction of useful information from a high dimensional dataset. These techniques not only increase the speed but potentially also the accuracy of many machine learning algorithms. There are almost countless articles written about selection methods reaching from manually, experienced based feature selection to completely automatic ways of selecting features. In this section we introduce some of the most used techniques, starting with the simplest one and moving on to more sophisticated algorithms afterwards.

Pearson Correlation In this method, we select features based on the Pearson correlation coefficient. More precisely, we calculate the absolute value of the correlation between each feature and the target and only keep the top n features based on this criterion.

Univariate Feature Selection Slightly more sophisticated, we can select the best features based on univariate statistical tests. We compare each feature to the target variable, to see if there is any statistically significant relationship between them. Such tests are referred to as analysis of variance (ANOVA). As already implicated by the word 'univariate' each feature gets analyzed separately, meaning when we analyze the relationship between one feature and the target variable, we ignore all other features. As a result, each feature has its own test score. Finally, we compare all test scores, and only select features with top scores [23]. In Python, the selectKbest() function implemented in the scikit-learn library [6] provides the statistically best features.

Recursive Feature Elimination (RFE) Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), we can use recursive feature elimination (RFE) by recursively considering smaller and smaller sets of features. First, we train the estimator using the initial set of features. The importance of each feature is obtained either through a coef_ or a feature_importances_ attribute. Afterwards, the least important ones are pruned from the current set of features. This procedure gets recursively repeated on the pruned set until the desired number of features to select is eventually reached [23].

Tree-based Feature Importance After training a tree-based method such as a Random Forest we can access the relative importance of each feature directly and use it for selection purposes. Random Forests provide two straightforward methods for feature selection, namely mean decrease impurity and mean decrease accuracy [2].

Boruta Last but not least, we want to introduce the Boruta algorithm, which is a smart algorithm to automatically perform feature selection. It was originally implemented in 2010 as a package for R by M. Kursa and R. Rudnicki [22] and adapted for Python by Daniel Homola later on. Using the Boruta algorithm, we compare all available features against a randomized version of themselves, referred to as shadow features. These features are created by random shuffling. To confirm or reject an original feature we use the binomial distribution.
Following the algorithm, we first fix a number of iterations to perform. For each iteration $1, \ldots, n$ we calculate tree-based feature importances by fitting for example a Random Forest. After comparing the importance of each original feature with the highest value recorded among all shadow features, we mark it as success or failure and compare the number of successes to a binomial distributed random variable in each iteration. Finally, we end up with three regions, namely the area of refusal, where features are considered as noise and get dropped, the area of irresolution, where features can be kept and the area of acceptance, where features are considered as predictive and should be kept [21].

When it comes to pure dimensionality reduction, another commonly used algorithm is Principal Component Analysis (PCA). Even though it could yield reasonable results it is strictly speaking not a feature selecting algorithm, because instead of selecting important ones it replaces the original variables with synthetic ones. Thus, we do not cover PCA in this thesis.

5.1.4. Normalization

Normalization can be a crucial part in machine learning as data sets often contain variables with different scales. The goal of normalization is to change the values of numeric columns in the data set to a common scale, without changing its structural behavior. Which normalization approach to use, or whether it is reasonable to normalize data at all depends on the situation. Nevertheless, we introduce some of the most common approaches.

Z-Score (Standardization): During this normalization process for each feature its mean is subtracted separately and afterwards the difference divided by the feature's standard deviation as follows:

$$x_i^{(Z_{\text{score}})} = \frac{x_i - \overline{x}}{\sqrt{\frac{1}{N}\sum_{j=1}^{N}(x_j - \overline{x})^2}},$$

where $\overline{x}$ represents the features mean,

$$\overline{x} = \frac{1}{N}\sum_{j=1}^{N} x_j.$$

This approach yields stationary features with mean 0 and a corresponding standard deviation of 1.

Min-Max Normalization: The second normalization method we present subtracts the minimum observed value from each feature and divides this difference it by the difference between the maximum and the minimum observed value of that feature, as follows:

$$x_i^{(MM)} = \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}.$$

This transformation yields rescaled data in the range $[0, 1]$.

Decimal Precision In the third normalization method, the decimal precision approach, we simply move the decimal point in each of the feature values. Hence, the transformed values are:

$$x_i^{(DP)} = \frac{x_i}{10^k},$$

where k is the integer that gives us the maximum value for $\left|x^{(DP)}\right| < 1$.

5.2. Predicting Price Movements

In this section, we present how to use LOB data to predict price movements, arguably one the most challenging tasks in financial markets analysis.
As a first step, we introduce two approaches to label target variables as three different classes. After that we briefly review the data pre-processing and last but not least we want to present the algorithms used and their performances.
In this task we investigate all four above mentioned stocks, but due to the tremendous amount of available data and computational restrictions we do not work with the whole data sets.

5.2.1. Target Labels

In term of price movements there are several potentially interesting targets. Although best ask and best bid must theoretically both be modeled separately, since the bid-ask spread is not constant [34], the so called market microstructure noise can be partially reduced by using midprices. Thus, in this thesis we focus on the midprice for more smoothed results in the long-run. We treat the problem of price movement prediction as a classification problem with three possible outcomes: up, down, and horizontally. We label these price movement as 1, -1 and 0.

Since we are only interested in significant price changes and want to reduce some noise we take only significant movements into account. We present two approaches to label the target:

Simple Approach: First we calculate the percentage change of the price as follows:

$$pct\_change_{t,t+1} = \frac{p_{t+1} - p_t}{p_t},$$

where p_t denotes the choosen (best ask, best bid, mid) price at time t. The label $l(t)$ for time t is defined as

$$l(t) := \begin{cases} 1 & , \quad \text{if } pct\_change_{t-1,t} \geq \alpha \\ 0 & , \quad \text{if } pct\_change_{t-1,t} \in (-\alpha, \alpha) \\ -1 & , \quad \text{if } pct\_change_{t-1,t} \leq -\alpha \end{cases},$$

where α determines the significance of the price movement.

Smoothed Approach: In order to reduce microstructure noise even further, we present an approach proposed by [31]. An averaging filter is applied over the future N values of the price to classify the changes in the price as follows:

$$l(t) := \begin{cases} 1 & , \quad \text{if } \frac{mp_t}{p_t} \geq (1+\alpha) \\ 0 & , \quad \text{if } \frac{mp_t}{p_t} - 1 \in (\alpha, \alpha) \\ -1 & , \quad \text{if } \frac{mp_t}{p_t} \leq (1+\alpha) \end{cases}$$

where p_t denotes the price at time t and $mp_t = \frac{1}{N}\sum_{i=1}^{N} p_{t+i}$ is the average of the future price events with window size $N = 5$. Again, α determines the significance of the price movement.
In both approaches the value of α can bee seen as a trade-off between the balance among classes and the meaningfulness of the produced labels. As α increases, so does the number of samples belonging to the no-movement class. Although raising this threshold would lead to a more balanced problem, meaning all three classes would contain about the same number of samples, high values of α are undesirable as they allow upwards and downwards movements to get categorized as no movement even though the movement could be significant.

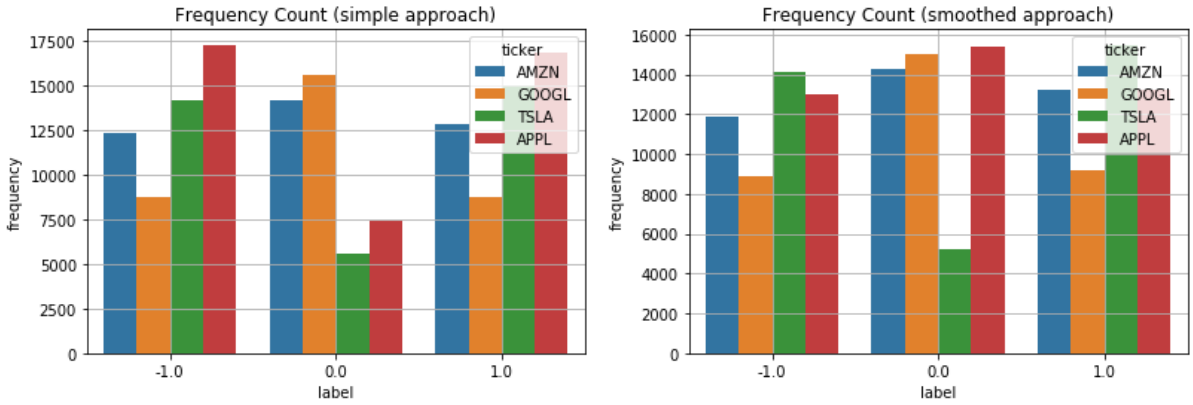
5.2.2. Pre-processing

As already indicated, due to computational limitations we can not use all available data but rather need to narrow it down to some parts. In particular, we want to only focus on the first five days of available data. Additionally we do not use every observation, but only every 70^{th} one for APPL and every 20^{th} one for AMZN, GOOGL, TSLA, respectively.
After loading the data we perform following steps for each data set:

1. First we specify our actual goal by labeling the targets to use, as described in section Section 5.2.1. Using $\alpha = 2 * 10^{-5}$ for the simple approach and $\alpha = 5 * 10^{-5}$ as well as $N = 5$ for the smoothed approach yields following results:

Table 5.4: Class partition

			Simple Approach			Smoothed Approach		
Ticker	**Price**	**Observations**	**1**	**0**	**-1**	**1**	**0**	**-1**
AMZN	Midprice	39322	12852	14160	12310	13226	14239	11857
APPL	Midprice	41618	16892	7421	17305	13200	15405	13013
GOOGL	Midprice	33112	8795	15593	8724	9178	15018	8916
TSLA	Midprice	34805	15066	5565	14174	15480	5201	14124



(a) Class frequencies (simple approach) (b) Class frequencies (smoothed approach)

Figure 5.1: Comparison of class frequencies

From Table 5.4 and Figure 5.1, which are summarizing the corresponding labels, we can clearly see that some targets have unbalanced classes. Thus, we need to use a weighted cross-entropy as loss function. We therefore modify the version of the cross-entropy defined in Section 2.2.2 to

$$L(\boldsymbol{y}, \boldsymbol{x}; \boldsymbol{\theta}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{K} w_k y_{i,k} \ln \hat{y}_{i,k},$$

where $K = 3$ represents the number of classes and $w_k = \frac{n}{K * \text{ frequence of class } k}$ are the class weights. This modification allows us to train the model with best generalization capability, by weighting all classes equally important.
In this thesis we focus on the more straightforward, simple approach. Besides the unbalanced classes, we can see that except for APPL the targets' distributions are very similar for both approaches. Given that observation and the fact that we are

interested in the pure predictive power of the LOB together with the capability of neural networks to capture the non-linearity in the LOB, it is sufficient to focus on the straightforward, simple approach. Nevertheless, to implement profitable trading strategies it might be reasonable to use the smoothed approach, as some noise gets canceled out there.

2. Next, we add the features presented in Section 5.1.2 to the data set. Extracting these features from the limit order book data already yield 111 dimensional input vectors. Remember, that one feature vector is extracted for every 70 or 20 limit order events that change the LOB. But, instead of using only the feature vectors extracted from the current time step, as originally proposed in [27], we propose extracting representations capable of capturing more temporal information as in [28]. Therefore we concatenate each consecutive 6 feature vectors together, yielding new 666-dimensional input vectors.

3. We select the most relevant features, by either using RFE to select 256 features or using Boruta for 50 iterations.

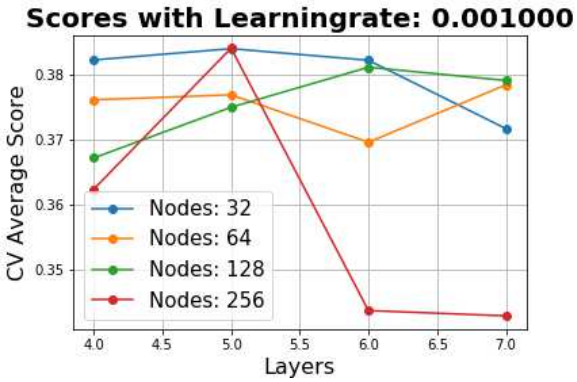
4. Last but not least, we normalize the input data, using the z-scores normalization approach, as described in Section 5.1.4.

5.2.3. Empirical Results

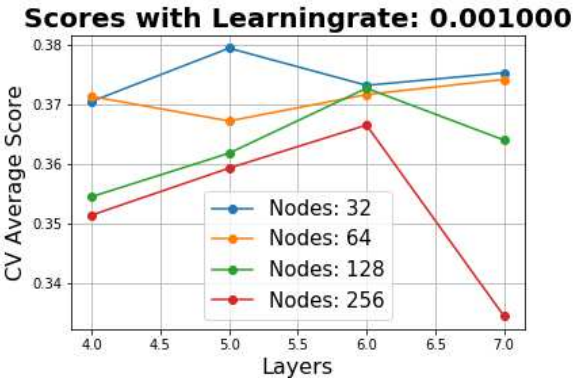
In this section we want to use empirical results to first compare different model architectures, using various hyper-parameter settings. Furthermore, we investigate the impact of different feature selection methods, namely RFE as well as the Boruta algorithm. Finally, we show that deep neural networks are indeed more capable of capturing the non-linearity in the limit order book, as indicated in [34], by comparing them to a single layer shallow neural network using the softmax activation function.
To quantify the comparison results we use accuracy as metric, giving the percentage of correctly identified observations.

Model Tuning Hyper-parameter tuning is another crucial part in machine learning projects, especially for more complex algorithms, such as deep neural networks. The tuning procedure consists of first, fitting many models with different parameter settings and evaluating each of them followed ba selecting the hyper-parameter setting that yields the best performing model.
For evaluation purposes we would usually split the available data into a train set to train the model and a test set to evaluate its performance, but in order to yield more reliable results we use a k-fold cross validation, as described in Section 2.4.1, instead. Due to the huge amount of available data it is sufficient to use a 3 fold cross validation in the experiment.
During the tuning process we mainly focus on the model architecture, meaning depth and width. Hence, we use the default value $batchsize = 32$, fix $epochs = 20$, select $tanh$ as activation function for the hidden layers and the $softmax$ function for the output

layer. Furthermore, we select the Adam algorithm from Section 2.3.2 as optimizer for all models.
To find the best model, we perform a grid search on $layers \times nodes\_per\_layer$ with $layers = \{4,\ 5,\ 6,\ 7\}$ and $nodes\_per\_layer = \{32,\ 64,\ 128,\ 256\}$. Since, we want to investigate the predictive power of deep neural networks, we only use models with at least four layers.
In order to further improve the results we realize the grid search for each learning rate in $learning\_rates = \{0.001, 0.005,\ 0.01,\ 0.05,\ 0.1,\ 0.5\}$. The tuning process can be implemented using the GridSearchCV() function from the scikit-learn Python library.
In Figures 5.2, 5.3, 5.4 and 5.5 we show the grid search results for each stock using both RFE and the Boruta algorithm. The presented figures are given in terms of average accuracy after realizing the 3-fold cross validation for each combination.
Note, that results are only presented for the best performing learning rates.

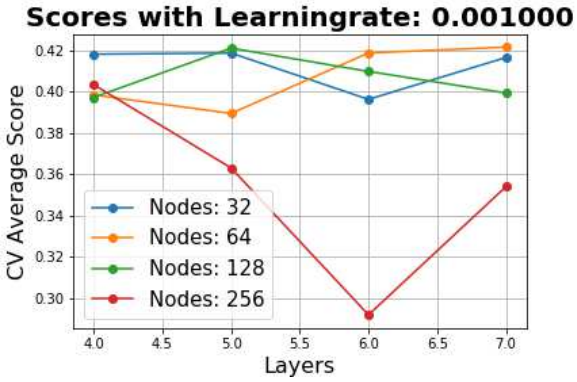


(a) AMZN, Boruta

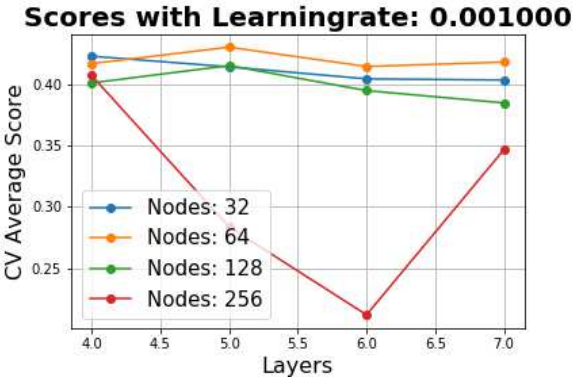


(b) AMZN, RFE

Figure 5.2: Grid search results for AMZN

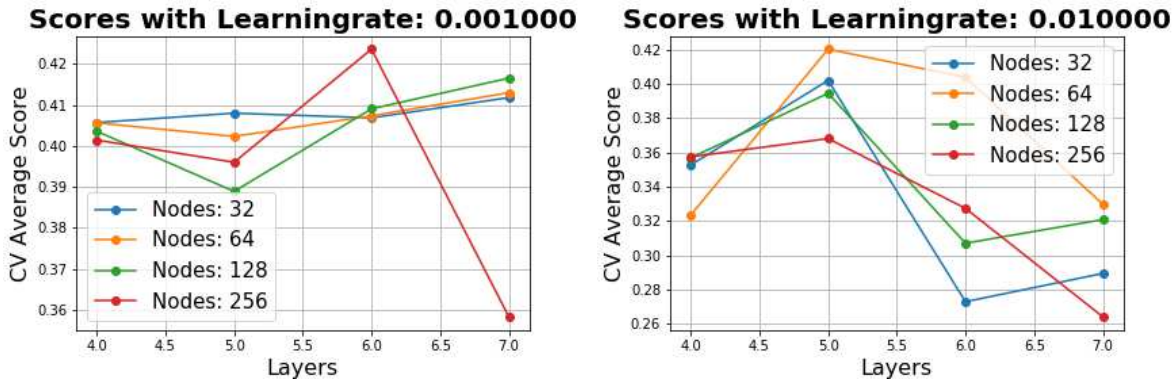


(a) APPL, Boruta



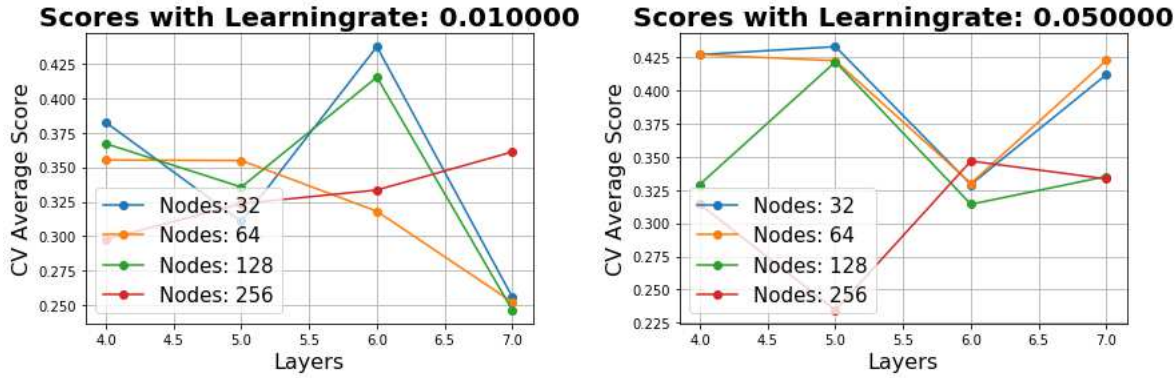
(b) APPL, RFE

Figure 5.3: Grid search results for APPL



(a) GOOGL, Boruta (b) GOOGL, RFE

Figure 5.4: Grid search results for GOOGL



(a) TSLA, Boruta (b) TSLA, RFE

Figure 5.5: Grid search results for TSLA

From Figures 5.2, 5.3, 5.4 and 5.5 we notice that in three out of four cases the Boruta algorithm outperformed RFE. Even though for APPL using the RFE algorithm yields slightly higher cross validation score than using the Boruto algorithm, we still prefer using Boruta, because it is much more stable regarding the cross validation score's standard deviation. In Table 5.5 we summarize the preferred hyper-parameter settings as well as the preferred feature selection algorithm for each data set.

Table 5.5: Preferred hyper-parameter settings and feature selection algorithm

Ticker	Learning Rate	Feature Selecting	Layers	Nodes per Layer
AMZN	0.001	Boruta	32	5
APPL	0.001	Boruta	64	7
GOOGL	0.001	Boruta	256	6
TSLA	0.01	Boruta	32	6

From Figures 5.2, 5.3, 5.4 and 5.5 and Table 5.5 we notice that it is beneficial to use just a few nodes, while a higher number of layers can increase the performance. Consequently, we prefer deeper instead of wider networks. Of course, this is neither a proof nor an universally valid rule just an observation for our available data.

Final Comparison Finally, we test the best performing multilayer perceptrons against a commonly used algorithm for classification, namely a multinomial logistic regression. In order to do that we first split the used data into a training set, containing randomly selected 80% of the available data, and a test set, containing the remaining 20%. Afterwards, we train the multilayer perceptron (MLP) only on the training set using the best hyper-parameter setting, as given in Table 5.5. As baseline model we use a multinomial logistic regression, by training a single layer perceptron (SLP) on the training data using the default hyper-parameter settings from keras. Additionally, we use the LogisticRegression() function from the scikit-learn package to fit an ultimate challenger model. In Table 5.6 we show the results of each fitted.

Table 5.6: Comparison of different algorithms

		SLP		**LogisticRegression()**		**MLP**	
Ticker	**Biggest Class**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**
AMZN	0.360	0.407	0.396	0.44	0.41	0.467	0.406
APPL	0.415	0.370	0.361	0.40	0.39	0.464	0.425
GOOGL	0.471	0.417	0.396	0.45	0.42	0.603	0.454
TSLA	0.433	0.413	0.404	0.43	0.41	0.477	0.430

We can clearly see, that the multilayer perceptron sometimes even by far outperformes the single layer one. But even more impressive is the fact that just tuning the model architecture (width and depth) together with using different learning rates is already sufficient to outperform the logistic regression model implemented in the scikit-learn package. Hence, we conclude that multilayer perceptrons are indeed able to capture parts of the non-linearity in limit order book data, making them far more suitable for predicting price movements than linear models, such as logistic regression. Moreover, outsourcing the training process to clouds system, such as AWS or AZURE, with by far more computational power, can improve perceptrons' accuracy even further. This would allow us to implement more sophisticated hyper-parameter tuning approaches.

5.3. Volatility Forecasting

In the second experiment we use neural networks to forecast volatility. Volatility it is a key parameter for pricing financial derivatives, as all modern option-pricing techniques rely on a volatility parameter. Furthermore, volatility is used in risk management and

portfolio optimization, which makes it, next to returns, one of the most important figures in the financial markets.

5.3.1. Pre-processing

Similar to the previous experiment we do not use all available data, due to computational limitations. This time we use data from every day, but only every 100^{th} observation for GOOGL and AMZN and every 150^{th} observation for APPL. Unfortunately we can not use TSLA, because the data-quality was insufficient for some days.
In order to prepare the data set for training purposes we perform following steps:

1. Again, we first specify the target variable. As discussed in Section 4 we use the two scales realized volatility estimator to approximate the realized variance and afterwards take the square root to end up with the estimated realized volatility. Normally we would do this procedure for each day, but since our longest available time series contains only 410 days, we split each day into 3 snapshots and work with intraday volatilities.

2. Next, we prepare the input data. We add the features presented in Section 5.1.2 for each observation. To keep the needed computational effort reasonable we summarize each snapshot with 8 equidistant feature vectors. After concatenating these 8 vectors and adding the last 8 intraday volatilities, we end up with 1008-dimensional input vectors. Thus, each vector contains the limit order book information from the previous snapshot together with the last 8 intraday volatilities.

3. Since using the Boruta algorithm does not yield any reasonable results here, we only use RFE and test it against using all features.

4. Once again, we normalize the input data, using the z-score normalization technique. To increase the model's performance, this time we also normalize the target variables. Although normalizing the targets is not a big deal, it is very important to mention that in order to receive correct predictions we have to use the inverse z-score transformation on the model's predictions.

5.3.2. Empirical Results

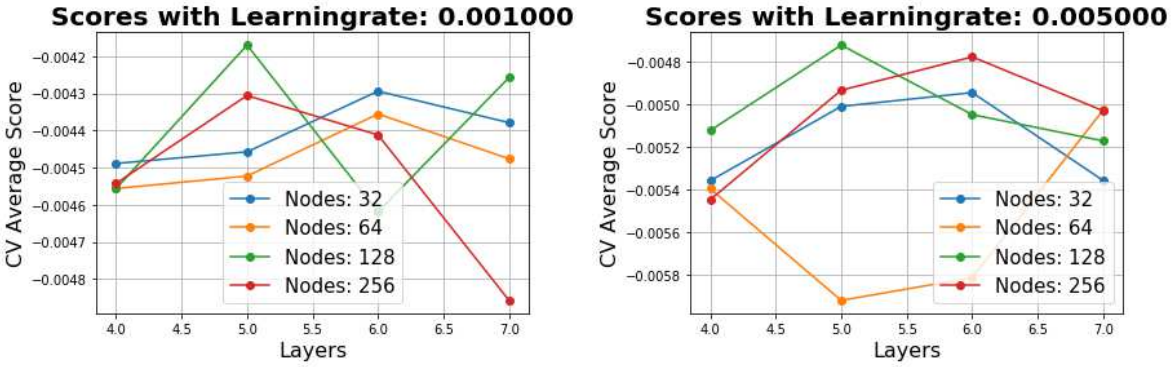
In this section we present the empirical results. Again, we start by comparing different model architectures and various hyper-parameter settings. For tuning purposes we use two different approaches, namely using the RFE algorithm for feature selection as well as using no feature selection algorithm at all. In the final test we compare the best perfoming deep neural networks to ARIMA models.
To quantify the results we use the root mean squared error (RMSE),

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|^2}$$

as well as the mean absolute percentage error (MAPE),

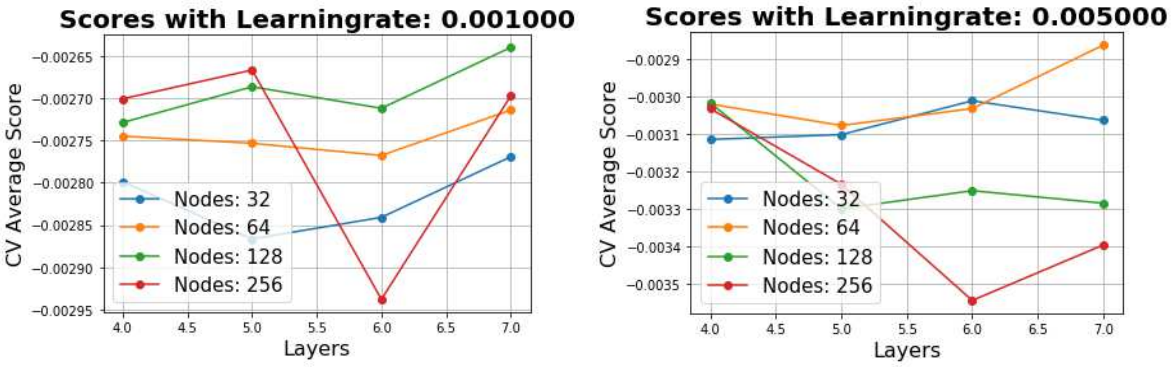
$$\frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right|.$$

Model Tuning Similar to the previous experiment, we perform a 3-fold cross validation to find the best hyper-parameter setting. Since we are mainly interested in the model architecture we fix $batchsize = 32$, $epochs = 20$ and select $tanh$ as activation function. For each learning rate in $learning\_rates = 0.0005, 0.001, 0.005, 0.01, 0.05$ we perform a grid search on $layers \times nodes\_per\_layer$, with $layers = \{4, 5, 6, 7\}$ and $nodes\_per\_layer = \{32, 64, 128, 256\}$. Furthermore, we perform the grid searches after using the RFE algorithm as well as for all features.
The results for the best learning rates are presented in Figures 5.6, 5.7 and 5.8. In order to stay consistent with the previous experiment, and therefore improve readability, the best results shall be on the top of the graphs. Therefore we choose the negative RMSE as error metric.



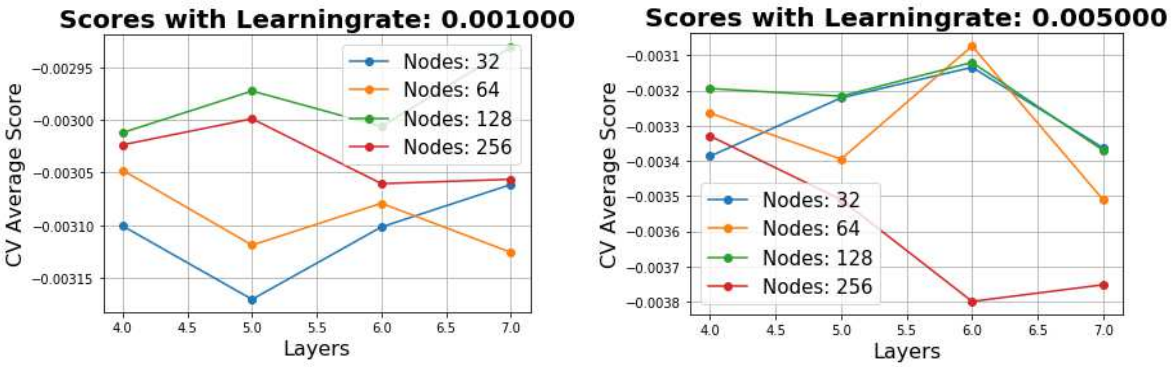
(a) AMZN, All Features (b) AMZN, RFE

Figure 5.6: Grid search results for AMZN



(a) APPL, All Features (b) APPL, RFE

Figure 5.7: Grid search results for APPL



(a) GOOGL, All Features

(b) GOOGL, RFE

Figure 5.8: Grid search results for GOOGL

In Table 5.7 we summarize the best performing hyper-parameter settings.

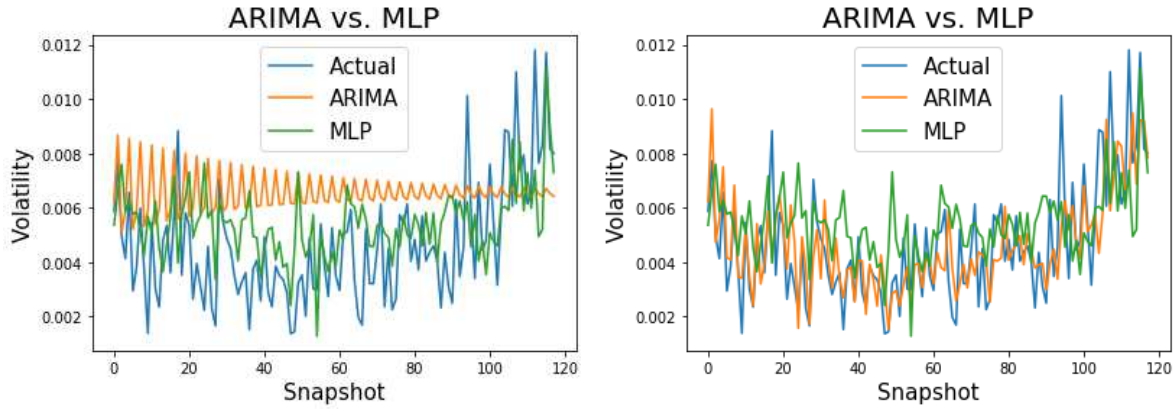
Table 5.7: Preferred hyper-parameter settings and feature selection algorithm

Ticker	Learning Rate	Feature Selecting	Layers	Nodes per Layer
AMZN	0.001	none	128	5
APPL	0.001	none	128	7
GOOGL	0.001	none	128	7

Surprisingly, despite the high number of features, we receive the best results without using any feature selection algorithms. Furthermore, we can not observe a clear trend to favor either wider or deeper networks.

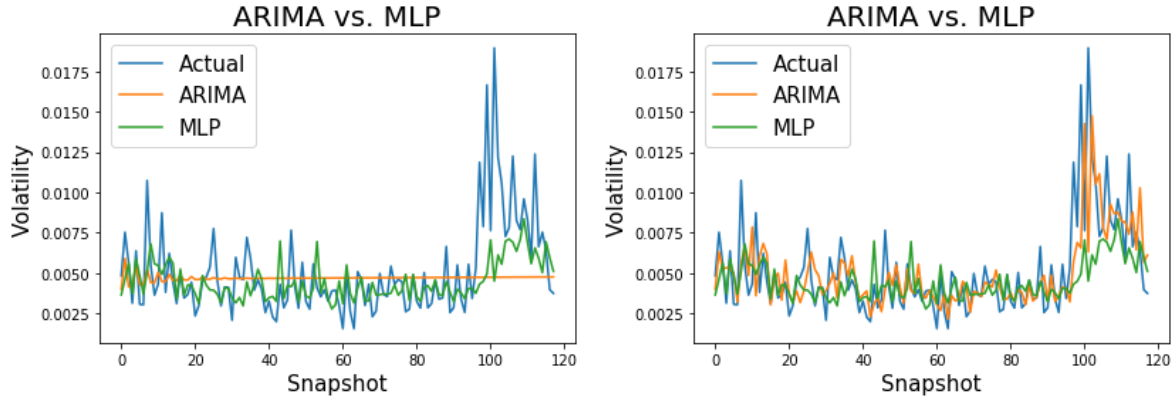
Final Comparison Finally, we do two different tests to compare the best performing multilayer perceptrons against ARIMA(p, d, q) models.
For the first test we simply use the least recent 90% of available data to train the neural network and calibrate the ARIMA model and the remaining 10% to test them. This test shows us how suitable the different models are for receiving good long-term predictions The second test starts similar to the first one, by training the neural network only once on the first 90% of the data and predicting the remaining 10%. But afterwards we do not only calibrate the ARIMA model once but in every time step. More precisely, we start with the same 90% of training data and then increase the training data one step at a time until we reach the most recent observation. In each step we calibrate a new ARIMA model and only predict the volatility exactly one step ahead. This allows us to increase the ARIMA model's precision but at the cost of computationally efficiency. Thus, this test compares a neural network's long-term predictions to short-term predictions of commonly used ARIMA models, which are calibrated with much more recent information.
In both tests we use the auto_arima() function from the pmdarima library in Python.

This function automatically finds and uses the best order, meaning the best figures for the parameters p, d and q.
In Figures 5.9, 5.10 and 5.11 we present the results for both tests. The plots on the left hand side represent the first test and the ones on the right hand side represent the second test.



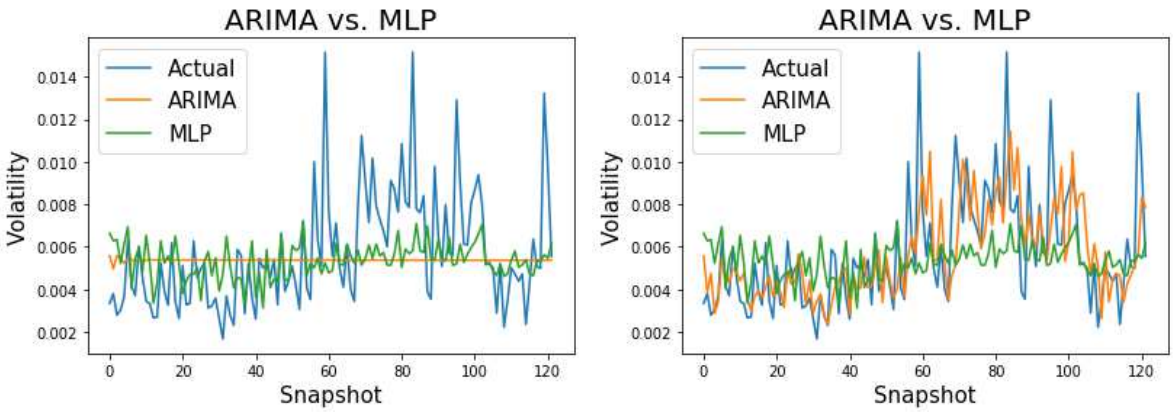
(a) without re-calibrating (b) with re-calibrating

Figure 5.9: ARIMA vs. MLP for AMZN



(a) without re-calibrating (b) with re-calibrating

Figure 5.10: ARIMA vs. MLP for APPL



(a) without re-calibrating (b) with re-calibrating

Figure 5.11: ARIMA vs. MLP for GOOGL

In Table 5.8 we present the MAPE corresponding to each plot above.

Table 5.8: Comparison of ARIMA model and MLP

Ticker	**ARIMA**	**ARIMA re-calibrated**	**MLP**
AMZN	77.097	26.921	51.607
APPL	39.375	28.227	27.760
GOOGL	39.079	25.918	35.349

Looking at the results clearly points out that on the one hand feedforward neural networks together with the limit order book information have a better performance when it comes to long-term predictions. But on the other hand in two out of three cases they are not able to outperform continuously re-calibrated ARIMA models. Thus, we are obviously dealing with a trade-off between precision and calibration effort.
However, since many transactions on the financial markets are very time critical it might not be feasibly to calibrate a model each time we want to predict future volatility. Thus, using pre trained neural networks are a reasonable alternative to classical ARIMA models to reduce calibration efforts and therefore increase speed.
Additionally it is important to notice that neural networks are performing best for huge data sets, whereas our biggest data set only contains 410 days. Thus increasing the amount of available data can yield a huge improvement of performance.

6. Conclusion

In this thesis we gave a detailed introduction into the theory of neural networks, where we presented basic architectures, particular deep feedforward neural networks. We not only described how to train them, but also focused on optimization and generalization techniques.
Furthermore, we presented a mathematically precise description of a limit order book as well as its actual data structure, followed by an unbiased approach to estimate realized volatility using noisy high-frequency data, referred to as two scales realized volatility estimator (TSRV).
Finally, we showed how to apply this methodology on real market data. The used limit order book data from the NASDAQ stock exchange was provided by the online tool LOBSTER. For each data set we used handcrafted features and investigated the impact of different feature selection methods, such as recursive feature selection and the Boruta algorithm to decrease training speed and improve performance. Afterwards we trained and optimized deep feedforward neural networks on the most important features to outperform commonly used linear algorithms for mid-price prediction such as multiclass logistic regression for all four stocks. Additionally, our proposed approach yielded better long term volatility forecasts than ARIMA models. Consequently, it reduces the necessity of re-calibration, which yields faster predictions and is therefore a potentially beneficial indicator for investors.
Nevertheless, predictions are always uncertain and investments are risky. A neural network in its basic form is neither capable of capturing the evolution of financial markets in the long term nor able to understand the financial risk of wrong prediction. Consequently, it would be an interesting future project to investigate recurrent models together with a strategy that adds some financial indicator as risk measure for trading purposes.

References

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer, 2018.

[2] Ando. *Selecting good features – Part III: random forests*. URL: https://blog.datadive.net/selecting-good-features-part-iii-random-forests/.

[3] Hans Bühler, Lukas Gonon, Josef Teichmann, and Ben Wood. *Deep Hedging*. 2018. URL: https://arxiv.org/pdf/1802.03042.pdf.

[4] Andriy Burkov. *The Hundred-Page Machine Learning Book*. Andriy Burkov, 2019.

[5] blog contributers. *Machine Learning Mastery*. URL: https://machinelearningmastery.com.

[6] scikit-learn developers. URL: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_classif.html#sklearn.feature_selection.f_classif.

[7] scikit-learn developers. URL: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html#sklearn.feature_selection.RFE.

[8] Joaquin Fernandez-Tapia. *Modeling, optimization and estimation for the on-line control of trading algorithms in limit-order markets*. URL: https://www.researchgate.net/publication/284900784_Modeling_optimization_and_estimation_for_the_on-line_control_of_trading_algorithms_in_limit-order_markets.

[9] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Vol. 1. Springer series in statistics New York, 2001. URL: https://web.stanford.edu/~hastie/ElemStatLearn/printings/ESLII%5C_print12.pdf.

[10] James Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*. Vol. 112. Springer, 2013. URL: http://www-bcf.usc.edu/~gareth/ISL/ISLR%20Seventh%20Printing.pdf.

[11] Xavier Glorot and Yoshua Bengio. *Understanding the difficulty of training deep feedforward neural networks*. 2010. URL: http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[13] Martin D Gould et al. *Limit order books*. 2013. URL: https://people.maths.ox.ac.uk/porterm/papers/gould-qf-final.pdf.

[14] Rainer Hirk. *Nichtparametrische Volatilitätsschätzer unter Market Microstructure Noise*. 2014. URL: http://repositum.tuwien.ac.at/obvutwhs/download/pdf/1635371?originalFilename=true.

[15] Tobias Hitzig. *High-frequency trading and limit order book indicators*. 2016.

[16] Kurt Hornik. *Approximation capabilities of multilayer feedforward networks*. 1991. URL: http://www.sciencedirect.com/science/article/pii/089360809190009T.

[17] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. *Multilayer feedforward networks are universal approximators*. 1989.

[18] Sergey Ioffe and Christian Szegedy. *Batch normalization: Accelerating deep network training by reducing internal covariate shift*. 2015. URL: https://arxiv.org/pdf/1502.03167.pdf.

[19] Pankaj K. Jain, Pawan Jain, and Thomas H. McInish. *The Predictive Power of Limit Order Book for Future Volatility, Trade Price, and Speed of Trading*. 2011. URL: https://ssrn.com/abstract=1787625.

[20] Diederik P Kingma and Jimmy Ba. *Adam: A method for stochastic optimization*. 2014.

[21] Dawid Kopczyk. *Feature Selection algorithms – Best Practice*. URL: https://dkopczyk.quantee.co.uk/feature-selection/.

[22] Miron B Kursa and Witold R Rudnicki. *Feature Selection with the Boruta Packag*. 2010. URL: https://www.jstatsoft.org/article/view/v036i11.

[23] Richard Liang. *Feature selection using Python for classification problems*. URL: https://towardsdatascience.com/feature-selection-using-python-for-classification-problem-b5f00a1c7028.

[24] Lobster. *LOBSTER data*. URL: https://lobsterdata.com/info/DataStructure.php.

[25] Lulu. *Lulu's Blog*. URL: https://lucidar.me/en/neural-networks/summary/.

[26] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. URL: https://doc.lagout.org/science/Artificial%20Intelligence/Machine%20learning/Machine%20Learning_%20A%20Pro%20babilistic%20Perspective%20%5BMurphy%202012-08-24%5D.pdf.

[27] Alec N.Kercheval and Yuan Zhang. *Modeling high-frequency limit order book dynamics with support vector machines*. 2013. URL: https://pdfs.semanticscholar.org/9e84/5fb8805509ad716c698c4cfa4f4f50a450fb.pdf?_ga=2.127074885.974364117.1583268636-1478772359.1583268636.

[28] Paraskevi Nousi et al. *Machine Learning for Forecasting Mid Price Movement using Limit Order Book Data*. 2019. URL: https://arxiv.org/pdf/1809.07861.pdf.

[29] Adamantios Ntakaris. *Mid-Price Movement Prediction in Limit Order Books Using Feature Engineering and Machine Learning*. 2019. URL: https://trepo.tuni.fi/bitstream/handle/10024/117394/978-952-03-1288-6.pdf?sequence=5&isAllowed=y.

[30] Adamantios Ntakaris, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. *Mid-price prediction based on machine learning methods with technical and quantitative indicators*. 2019. URL: https://arxiv.org/pdf/1907.09452.pdf.

[31] Adamantios Ntakaris, Giorgio Mirone, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. *Feature engineering for mid-price prediction with deep learning*. 2019.

[32] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. URL: http://arxiv.org/abs/1609.04747.

[33] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. 1985. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a164453.pdf.

[34] Justin A. Sirignano. *Deep Learning for Limit Order Books*. 2016. URL: https://arxiv.org/pdf/1601.01987.pdf.

[35] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. *On Early Stopping in Gradient Descent Learning*. 2007.

[36] Lan Zhang, Per A Mykland, and Yacine Ait-Sahalia. *A tale of two time scales: Determining integrated volatility with noisy high-frequency data*. 2005. URL: http://wwwf.imperial.ac.uk/~pavl/AitSahalia2005.pdf.