Prediction of financial time series with artificial neural networks to recognize upcoming trends

Build an autonomic trading algorithm to beat the finance market

Bachelor Thesis

for the

Bachelor of Science

at Course of Studies Applied Computer Science

at the Cooperative State University Stuttgart

by

Marius Herget

4. September 2017

Die Verschwiegenheitsklausel der IBM Deutschland GmbH wurde 26.02.2019 aufgehoben.

Time of Project	June - September 2017
Student ID, Course	6542702, TINF14A
Company	IBM Deutschland GmbH, *Stuttgart, GER*
Supervisor in the Company	Christian Bernecker
Reviewer	Karl Friedrich Gebhardt

Author's declaration

Unless otherwise indicated in the text or references, or acknowledged above, this thesis with the topic:

Prediction of financial time series with artificial neural networks to recognize upcoming trends

Build an autonomic trading algorithm to beat the finance market

is entirely the product of my own scholarly work. This thesis has not been submitted either in whole or part, for a degree at this or any other university or institution. This is to certify that the printed version is equivalent to the submitted electronic one.

München, 4. September 2017

_____

Marius Herget

Selbstständigkeitserklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema:

Prediction of financial time series with artificial neural networks to recognize upcoming trends

Build an autonomic trading algorithm to beat the finance market

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

München, 4. September 2017

_____

Marius Herget

Zusammenfassung

Das übergreifende Ziel dieser Bachelor Thesis ist die Evaluierung und Entwicklung einer Analyseprozedur zur Vorhersage von Zeitreihen. Der Fokus liegt dabei bei der Bestimmung von Qualität, Typ und bester Vorhersagemethodik. Zusätzlich werden allgemeine Methoden zur Vorhersage dieser Zeitreihen vorgestellt.

Dafür werden verschiedene Möglichkeiten der Zukunftsprognose verglichen und ein allgemeiner Ansatz zur Vorhersage vertieft (neuronale Netze). Möglichkeiten der Konfigurationen sind Methodik (z. B. Initialisierung, Lernalgorithmen) und Topologien (z. B. Anzahl von Eingangs-, Versteckten- und Ausgangsneuronen, Aktivierungsfunktionen).

Im praktischen Abschnitt wird ein grundlegender Prototyp für die Prognose von Zeitreihen mithilfe von Python und Tensorflow implementiert. Nach einem Vergleich von verschiedenen Frameworks wird die detaillierte Codebasis erläutert und einige Testergebnisse präsentiert.

Durch eine Reihe von Experimenten wird dabei eine optimale Netzwerkkonfiguration für das Beispiel des Mini Dow Jones ermittelt. Dabei wird eine maximale Genauigkeit von 66% für Vorhersagen über die Steigung/Abnahme der Zeitreihe in den nächsten 10 Sekunden erreicht.

Abstract

The overall ambition of this bachelor thesis is to evaluate and develop an analysis procedure which is able to valuate time series. The focus is on measuring the quality, type and best prediction method with a learning algorithm. In advance, general methods to predict those time series are presented.

Therefore, various capabilities of prediction will be compared and result in a universal approach to prediction (neural network). Possible configurations of the forecasts are the methodology (*e.g.* initialization, learning algorithms) and topology (*e.g.* amount of input/hidden/output neurons, activation functions).

In the practical part, a basic prototype for predictions of time series based on Python and Tensorflow is implemented. After evaluating and comparing different frameworks, the detailed code base is explained and some test results are presented.

By accomplishing a series of experiments an optimized configuration for a neural network based on the Mini Dow Jones is explored. The model reaches a maximum accuracy of 66% for predicting short-term trends (rising/falling) within the next 10 seconds.

Contents

List of Figures

List of Tables

Listings

Acronyms

ANN Artficial Neural Network.
API Application Programming Interface.
ARIMA Autoregressive Integrated Moving Average.
CF Cashflow.
CNN Convolutional Neural Network.
DAX Deutscher Aktienindex.
DB Database.
DHBW Duale Hochschule Baden-Württemberg.
EANN Elman Artficial Neural Network.
EU European Union.
FANN Feedforward Artficial Neural Network.
GA Grand average.
HP Hewlett Packard.
HPFS High Performance File System.
HTML Hypertext Markup Language.
IBM International Business Machines Corporation.
IoT Internet of things.
IV Intrinsic Value.
JSON JavaScript Object Notation.
MA Moving average.
NASDAQ National Association of Securities Dealers Automated Quotations.
NN Neural Network.
PU Processing Unit.
S&P Standard & Poor's.
SARIMA Seasonal Autoregressive Integrated Moving Average.
SI Seasonal index.
SQL Structured Query Language.
TDNN Time Delay Neural Network.
TS Time series.
TSC Technical Steering Committee.

1. Introduction

Forecasting the future and traveling back in time has always been one of the mankind's most imagined dreams. While traveling in time is up to today physically impossible, predicting future events come more and more to the science' at hand. Using massive information collections called *Big Data* companies are *e.g.* already able to make a guess what a customer wants to buy in the near future. With more analysis methods and tools, combined with even a larger set of Big Data, other predictions examples seem to be within mankind's reach. Another example of forecasts are playlists from different music streaming providers. Those are so advanced, that automatic functions to add new songs to a playlist are possible. There are even more application areas than customer service. In 2016 Donald Trump was elected to the president of the United States of America. Several rumors claimed that this was only possible by micro targeting individual voters during his campaign. With the use of Big Data, he and his campaign organization could have been able to perfect his speeches and approach undecided voters with delegates.

This thesis tries to describe, analyze and apply a certain mechanism of this methods: Neural Networks. The financial world is one of the most discussed topics in the world. A lot of people wants to be rich and earn money just by spending money in the right investment at the right time. By predicting a financial market segment, this dream could come true quite easily. Furthermore, financial time series are very complex and contain most issues forecasting can cause.

Instead of naming both genders, this thesis will use always the male personal pronoun if it is referencing to a not further known person.

1.1. Problem

Instead of trying to predict the world's future, a much smaller problem space is addressed: time series. In more detail, financial time series of different types. The goal is to create a system which can predict upcoming trends in a short-time window. In the past, this was seen as impossible, since markets are influenced by an unknown amount of mostly unknown factors. With the upcoming new technologies and larger sets of real-time information, these factors are now identifiable and observable.

This thesis should give a detailed overview of current methods for forecasting. Furthermore, it examines whether it is possible to predict a financial time series. Therefore, the optimal configuration needs to be examined and applied to one market segment.

The higher target is to build an automatic trading application which is precise enough to beat the market. Therefore, a prototype is implemented which uses raw data to predict upcoming trends and values with a neural network.

1.2. Motivation

The main motivation of this thesis is to gain knowledge about neural networks. This knowledge should evaluate if it is possible to apply NNs to time series within the support business to increase customer satisfaction. In a real-life scenario, for example, a sentiment of a support conversation can be analyzed to predict if an escalation or special treatment of a ticket needs to be done. Since there is no support data available, a financial time series is used.

Another factor is to evaluate how complex, cost-intensive and resource demanding systems are to predict time series. This thesis is using financial time series since these are very detailed and contain a lot of information. Those are the perfect test objects to receive confident results.

1.3. Structure

The thesis is structured in a theoretical and practical part. After this introduction, there are insights to various topics starting with the general financial market in chapter 2. After an overall presentation of time series and some methods to analyze them, different market theories are annotated. The theoretical chapter closes with an introduction to neural networks and the current state of the art.

The practical part starts with chapter 3 and is carried on in chapter 4. It evaluates and implements a prototype to solve the described problem. Different deep learning frameworks are evaluated and an implementation architecture is constructed. After describing the detailed implementation, the results are compared and evaluated in chapter 5. In the end, chapter 6 summarizes the results, gives an overview to the future directions of the project and discusses the success or failure of this thesis project.

2. Theoretical background

Since the first predecessors of stock markets evolved in the 13th century, the overall complexity and ambitions raised over the years to one of the most complex markets in the world [55].

Although the term of *bourse* (exchange) derived from the Dutch **Huis ter Beurze** in Bruges the first official stock exchange took place in the 1600s when the Dutch East India Company issued bonds and shares to the general public. Other predecessors like the *Belgium* based meetings of brokers and money lenders, are not considered as stock markets since there was no usage of bonds and stocks. [8, 5]

Today there is a huge amount of bourses located all over the world whereby sixteen have a market capitalization of over $\$1T$. The biggest stock exchanges by market capitalization are the *New York Stock Exchange* (18.486 trillion USD), *NASDAQ* (7.449 trillion USD) and the *Japan Exchange Group* (4.91 trillion USD) [70].

In the following chapter, the basics of the finance market and used representation methods are described. Furthermore, the current state of analysis techniques and the fundamental concept of neural networks will be discussed.

2.1. Introduction to the financial world

The scope of this thesis are random walks within the financial system. A random walk is a value whose future data (*steps* or *direction*) cannot be anticipated based on its historical knowledge [56, p.24]. It is often represented as a graph. In the scope of *finance markets* it is intended as the not predictable short-terms changes in prices [56, p.24].

To understand which random walks exist in the international market and how they should be interpreted, it is important to describe the overall concept, the different market segments, and the already existing approaches to interpret those.

Financemarket

The term *finance market* is broadly considered as "markets that deal with cash flows over time, where the savings of lenders are allocated to the financing needs of borrowers. Financial markets are composed of money markets and capital market" [53, pp.114-130]. Another widely spread definition is: "The financial market is a broad term describing any marketplace where trading of securities including equities, bonds, currencies, and derivatives occurs" [80]. Furthermore, the term *asset* will be used later to describe different investing possibilities (instruments). Lee and Lee [53, p.226] differ between two kinds of assets:

- "Real assets are land, buildings, and equipment that are used to produce goods and services."
- "Financial assets are claims such as securities to the income generated by real assets."

This thesis is based on their definition of the *financial assets* while using the term *assets*.

Market segments

The international financial market consists of different sectors where different kinds of assets are traded. Moreover, there are two legal types of markets within the EU:

Regulated market In Germany, the *regulated market* is a coordinated place which is further defined in article 2, paragraph 5 of the Securities Trading Act. More precisely, this means that the German and in some cases EU law control and regulate the market. All participants are required to have a permission assigned after an approval process. Two example requirements for a company are: at least three years old and an estimated value of the shares of 1.25 million euros. [40]

Regulated unofficial market In comparison to the regulated market, the regulated unofficial market's rules are given by the corresponding bourse. These can vary from location and country. As an example, the *Terms and Conditions Regulated Unofficial Market* for the regulated unofficial market of the *Börse Stuttgart* can be viewed at [36].

Another way to differentiate the market segments is to distinguish their tradable goods (assets) also called *security*.

A security is a tradable asset representing an ownership of a "a publicly-traded corporation (via stock), a creditor relationship with a governmental body or a corporation (represented by owning that entity's bond), or rights to ownership as represented by an option" [84]. Example for securities are stocks, government bonds, mortgage bonds, public-sector bonds, corporate bonds or derivatives. In general it can be distinguish between *equity* and *debt* security. An equity security represents an ownership interest in an actual entity in form of shares of the capital stock. Although securities often pay out dividends (profit sharing), normally those are not entitled to regular payments. An equity security does entitle the holder to the right of co-determination. In comparison, a debt security represent worth which is lend, need to be paid off in certain amount of time. [84]

Stocks *or Share* A *stock* is a security which represents a partial holding of a company in percentage of the total shares available. Stocks have an important role in the economy like raising capital for businesses, facilitate company growth or as an indicator for the current state of the economic system [90, pp.9ff]. A detailed overview of the stock market literature can be viewed in [90].

Indices An *index* is a way to classify a representative group of individual data points and to measure changes statistically within it. The result can be used to make comparisons of data across time. Therefore a base value is chosen which is usually 100 and other data segments are contrasted to this base. [72, 73]

Futures A *future (contract)* is a financial commitment to buy or sell an asset or good at a scheduled date and determined price. Those are used to speculate on the price movements. Therefore, the buyer can buy an asset at a future date to sell

it afterward. A seller otherwise is sure to sell his good for a certain guaranteed price. [81, 61]

Options An *option* is a contract between an *option writer* and an *option holder* to trade an asset at an agreed-upon price during a certain time period. The option writer has the right but not the obligation to buy it. This allows the buyer to speculate while the seller has a lower risk of holding an asset. [83]

Although reviewing a large amount of fundamental literature for the financial market, there are no general definitions of the terms mentioned above.

Investments and speculations

Moreover, there are a lot of interpretations of the term *investing*. Malkiel [56, p.26] suggests it as an approach to "gain profit in the form of reasonably predictable income (dividends, interest, or rentals) and/or appreciation over long term". Graham [38, pp.18f] defines it as a contrast to the term *speculator*. Therefore, he supplements his definition from *Security Analysis: The Classic 1934 Edition* of differentiating the discrepancy between the terms. Accordingly, Graham characterizes the operation of an investment as a "through analysis promises safety of principal and an adequate return" founded action [38, pp.18f]. All other activities are speculative according to him.

In the definition of investing there are multiple mentions of the term *speculator* (*speculation*). Thus, there is already a description from Graham [38, pp.18f] not all other authors define their interpretation explicit. Malkiel [56, p.26] differentiated it over the time period for the investment return. He adds the aspect that the predictability of the return also discerns a speculation. Lambin [52, p.14] defines it as the "practice of engaging in risky financial transactions in an attempt to profit from short- or medium-term fluctuations on the market value of a tradable good, rather than attempting to profit from the underlying financial attributes embodied in the tradable good, such as capital gains, interest or dividends".

To sum up all mentioned authors distinguish an *investment* and a *speculation* in some aspects summarized in table 2.2. The definitions this thesis is using are:

Investment	Speculation
Long-term	Short-term
Based on fundamental factors which are kind of predictable	Resting on *soft facts* like market psychology, assumptions or rumors
Low or moderate risk	High risk
Anticipated small and steady value of return	Anticipated high and sporadic rate of return
Purchasing an asset with a longtime enhancement in value	Doing a risky financial action in hope for a substantial revenue

Table 2.2.: Investment compared to speculation

Investment An *investor* is somebody spending a considerable amount of worth in a long-term asset-based on predicted facts. The returning income is assumed to be moderate and steady.

Speculation A *speculator* is somebody supplying a considerable amount of worth in short-term assets knowing the high risk of the transactions. The justification is based on a non-predictable theory of environmental sentiments or non-provable assumptions. The revenue is supposed to be a substantial increase of the spent wealth. As Graham already pointed out, a speculator wants to beat the market.

Return of investment and predictability

As already mentioned in the introduction of this chapter the main focus of this thesis are *random walks*. A return of investment in such an environment depends on future events and a good, steady income from it rely on the investor's ability to predict this future [56, p.28]. Moreover, the income can be defined at any time t as [82]:

$$Profit(t) = \frac{AssetValue(t) - InvestedWorth(t_{purchase}) - \sum_{i=t_{purchase}}^{t} Transactioncosts(i)}{InvestedWorth(t_{purchase})} \tag{2.1.1}$$

This equation means that the profit at time t is the current value of the investment minus the costs when buying it ($InvestedWorth$) and the sum of all transaction costs to this momentum. It outputs the percentage increase of the asset. Furthermore, for a

long-time purchase the inflation needs to be considered as well [56, pp.26f]. The latest inflation in Germany was 0.5% in 2016. Thus, the average inflationrate from 2008 to 2016 was approximately 1.26% [13]. Moreover, CESifo-Gruppe [13] predicts that the future increase rise to 1.8% in 2017 and 1.7% in 2018. Accordingly, the European Central Bank [26] inflation rate goal is a rate below but close to 2%. Therefore, the overall profit should excides this percentage. Equation (2.1.1) now can be supplemented with the inflationrate r [82]. In addition, the overall market will grow as well. So if someone wants to beat the market, he has to perform better than the overall market. Therefore, the market growth rate m is also added to the equation.

$$Profit_{rm}(t) = \frac{(1 + Profit(t))}{(1 + r + m)} - 1 \tag{2.1.2}$$

For example, an investor buys assets of an DAX company worth of 50000€ at the first of Januar 2016 with an purchase cost of 1000€. At the end of 2016 (31.12.2016) he sells those for 65000€:

$$Profit(1483171200000) = \frac{65000€ - 50000€ - (1000€)}{50000} = 0.28 \tag{2.1.3}$$

$$Profit_{rm}(1483171200000) = \frac{(1 + 0.28)}{(1 + 0.005 + 0.068)} - 1 \approx 0.1929 \tag{2.1.4}$$

The profit is calculated at $31.12.2016$ at 9am (in javascript milliseconds format: 1483171200000). Like already stated out, the inflation rate in Germany was 0.5%. The DAX rises from end of 2015 to the end of 2016 with approximately 6.8% [12]. The eq. (2.1.3) shows, that the investor performed way worse if he/she considers the inflationrate of currency and the overall market growth.

A detailed view, on how inflation rates can be calculated and how to handle other aspects of an asset return can be found in "Asset returns and inflation" [29].

To understand how the market participants try to predict the sector, it is important to explain the two main basic approaches [25, pp.526f]:

- **Charting** Discover patterns in visual representations like charts.
- **Market technique** Calculating of trends (signals) to purchase or sell based on mathematical equations.

Charting is one of the most common concepts within the modern financial market. Users assume that historical data of the time series implicit information to predict its future value. [78, p.164]

A more specific approach to the financial market is the *efficient-market hypothesis* by Fama [28] in 1970. His theory is based on the idea that an ideal market is one "in which prices provide accurate signals for resource alloation" [28, p.383]. Furthermore, Fama declares, that market participants can make decisions under the assumption that asset prices at any time *fully reflects* all available information in such a market. He added some conditions for an efficient market [28, p.387]:

- No transaction costs
- All information are costless accessible for all market participants
- All participants admit on an implication of current information for a current price

If a market fulfills these aspects, a price of an asset *fully reflects* the available intelligence. More information and an empirical approach to the topic can be read in [28].

In contrast to the *fully reflected prices* the second approach is that prices are not also influenced by external or unknown aspects. Therefore, in this model fluctuations are the result of external phenomena such as political decisions, external rumors or environmental events. [78, p.164]

Current and classic methods how the financial market tries to predict the future directions of a random walk are discussed in sections 2.2 and 2.3.

Candlesticks

The *Candlestick* chart concept is rooted in the 17th and 18th century in Japan at the Osaka bourse for rice prices. [96, p.1180, 25, p.527, 64, pp.105f, 37]. A candle is one element within the chart representing a part of the random walk within the time t. One unit is based on four values: *Opening*, *High*, *Low* and *Closing*. Figure 2.1 illustrates the two types of candles with the corresponding mathematically expressions [91, p.264]. The given data S_{t_o} and S_{t_c} forms a cell body based on the follwoing definition:

- If the opening is below the closing value, the cell body is white (or sometimes red). Therefore, the abstracted graph is overall increasing within the time t. Otherwise, the cell body is black and represents an overall decreasing segment [37].
- The maximum of the abstracted passage is the highest point of the wick (also called *upper shadow*). Accordingly, the minimum forms the *lower shadow* [96, p.1180].

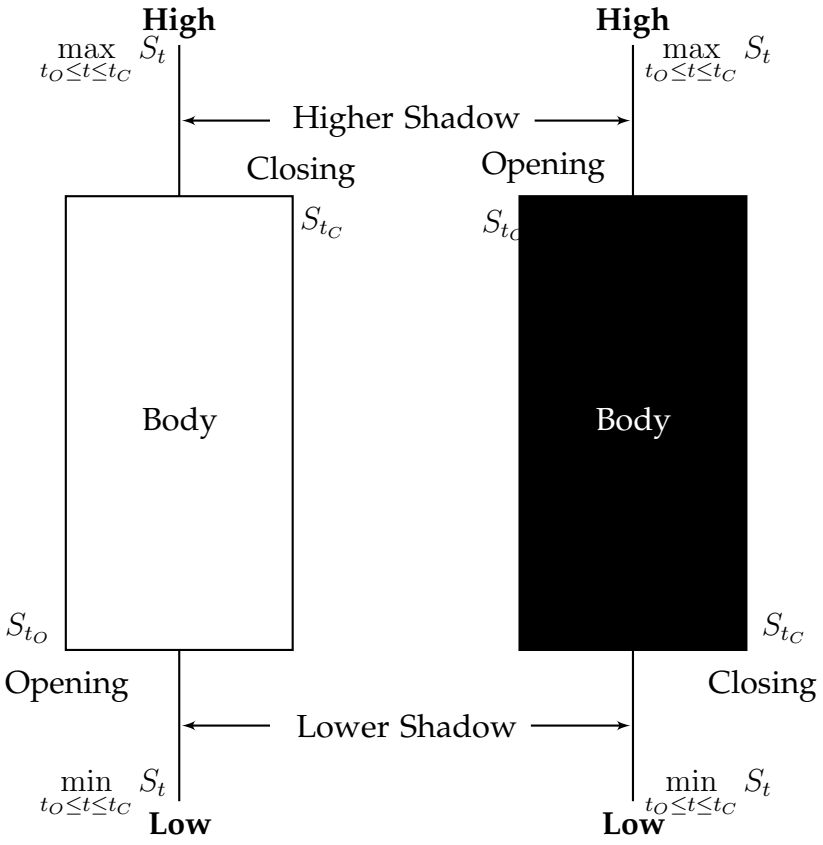


Figure 2.1.: Schemas of basic candlesticks with labels [37, Abb. 1]

Figure 2.1 shows the resulting schemas of the two candlesticks with the mathematical expressions [91, p.264]. Further information in different contexts are given in the work of Valeev and Terpugov [91] and Xie, Fan, and Wang [96].

2.2. Time series

Palit and Popovic [67] defines a *time series* as a "time-ordered sequence of observation values of a physical or financial variable made at equally spaces time intervals" [67, Chapter 2.1].

The goal of this section is to give a brief overview of how to examine a *time series (TS)*. The idea, thereby, is to use mathematical models to describe its behavior. It is sometimes possible to derive predictions based on those models. If an exact forecast calculation is possible it would be entirely *deterministic* but often there is an unknown amount of unknown factors which influence the TS. Nevertheless, it is sometimes possible to derive a design which can be used to get the *probability* of a future value in a specific time period. This kind of model is called *stochastic model*. [7, p.6]

The tactic of exploring a time series is to separate it into different steps: first, observations about a series need to be detected and developed in characteristics. Afterwards, a model is built to abstract the TS. Therefrom, predictions can be made via analysis and in combination with the characteristics, an attempt to control the behavior can be considered [92, p.225]. To achieve this, four components are usually examined: (i) *trends* (or long-term movements), (ii) *cycles* (or fluctuations), (iii) *time depending movements* (or seasons) and (iv) *random/stochastic* movements [92, p.226].

Moreover, there are several types of time series which will be examined in the next section. Afterwards, the different modeling methods are described. The third section discusses how the described characteristics and methods can be used to analyze a TS.

2.2.1. Types

A type of a time series is depending on its following charasteristics [67]:

- stationarity
- linearity
- trend
- seasonality

The *stationarity* factor can be seperated in two classes. A stationary time series has a *statistical equilibrium* with provabilistic properties like the *constant mean level* or *constant variance* do not change over time. The other class, a *non-stationary* TS, has varying properties which are sometimes staged as exponentially weighted moving averages. [7, p.7]

The second characteristic *linearity* describes whether a time series is *liner* or *non-linear*. A linear TS follows a specific direction which can be approximated by a linear function ($f(x) = mx + b$). A nonlinear one does not have this function and follows various directions. A series can change their linearity in different time periods.

Seasonality describes a periodic, recurring event within a time series. If a TS is season depending, forecasts can be made based on the time period. [3]

2.2.2. Modeling

There are different approaches for modeling a series. *Box-Jenkins* is a stochastic model building strategy to determine the most efficient stochastic model which fits the TS. In more detail it involves four related and sequential stages: (i) classify the time series (ii) consider the most efficient parameters for the model of the identified class (iii) test the efficiency of the model via diagnostic checks and (iv) predict the future behavior of the series with the model [7, pp.16,193, 92, p.230].

Another way to model time series is the *seasonality model*. As already evident from the name it only considers one characteristic of a time series. By finding a pattern that repeats for each period it is able to make time dependent forecasts. Therefore, a period is concrete defined time interval which occurs in a certain sequence. This can be an annual circle which is 12 periods long or a segmentation in quarters (4 periods) [3]. Firstly, this method detects a *seasonal index (SI)* which compares the average of one period to a *grand average (GA)*. The grand average is often the overall mean of all values within the data set whereby a SI of 1.00 in an annual pattern is the $\frac{1}{12}$ of the GA. The second step is to *deseasonalizing* the data. In this adjustment process, recurrent and periodic variations over a short time frame are removed. This is achieved by simply dividing each time series observation by the corresponding seasonal index. The final step is forecasting new data based on the model by using a seasonal factor in combination with the current trend of the timeseries [3].

The *random walk* model's approach is that the "most recent observation is best guide to immediate next prediction" [1, p.1442]. This concludes that all important information data for a prediction is included in the latest data set. The method, also known as the

Naïve model, is widely used in the financial market [1, pp.1442f] although it is only capable of predicting distinct trends with few fluctuations [21, p.10].

Another famous stochastic process is the *Markov chain* resulting in the *Markov property* [9, Preface p.IX]. This poverty is basically just a statement that defines whether it is possible to make predictions for the future only based on its present status in comparison to its full history. This leads to the theorem, that the future of a process is independent of the greater past and only depending on its recent values within a Markov chain. Those chains are often referenced as "collection[s] of random variables" based on a "countable time-set" [60, p.3]. The mathematical background can be read in [65, 60, 34]. In the context of time series, hidden Markov models can be applied. Since these would exceeded the scope of this thesis, further information can be found in *Hidden Markov Models for Time Series - An Introduction Using R, Second Edition*[98].

2.2.3. Analysis

This section describes some of the commonly used time series analysis methods. More information on the specific methods can be found in the different literature.

One method to identify trends is the *moving average (MA)* which uses successive segments of the time series to create an abstraction. Therefore, different data points at specific time spots are used to calculate an arithmetic mean. The goal is to eliminate season depending movements and fluctuations [92, p.225].

Another analysis method is *regression* whereby a dataset is used to recognize a relationship "of one variate on another in actual quantitative terms in contrast to correlation" [71, p.1661]. In more detail, the most popular version is the linear regression whereby the relationship is assumed to be linear [27, p.19].

The third method is *autocorrelation* process which relates the observations of a TS over time to each other. Therefore, it measures previous data set's influences on the current value by the autoregression coefficient ϕ [92, p.231]. This is an extension of the moving average method which is further examined in [92, pp.231ff].

The *autoregressive integrated moving average (ARIMA)* method combines two of the most famous methods to analyze time series into a general model for time series. It uses the auto-regressive elements to represent the lingering effects of previous scores and the integrated element representing the trends present in the data. In addition, it uses the MA to eliminate random shocks in the data set. The overall attempt is to filter out high-frequency noise in the data to detect local trends. It can be extended to the *seasonal autoregressive integrated moving average (SARIMA)* which acknowledges seasonal components. Two drawbacks of the ARIMA model are that it assumes linear relationships between independent and dependent variables in the data set. It also assumes a constant standard deviation in errors in the model over time. [95]

2.3. Market theories and trend analysis

Besides stochastic models, there are some other methods of how to interact in the financial market. This section describes some techniques to analyze and beat the market.

2.3.1. Firm-Foundation Theory

This theory was introduced by different authors but it is often associated with Williams [93] who designed the classical concept. The principle of the *Firm-Foundation* concept is, that every asset has an *instrinsic value (IV)* characterized as a "firm anchor [...] which can be determined by careful analysis of present conditions and future prospects" [56, p.29]. Williams [93, p.29] uses the dividend income to calculate that anchor. Moreover, he establishes the idea of *discounting* into the process. This concept introduces another aspect at the profit calculation similar to the in section 2.1 mentioned inflation rate. The author proposes to look at the income backward and anticipates the future worth of today's income [56, p.29]. On the other side, the investment company Cook & Bynum [15] summarize it in their own equation:

$$\textit{Intrinsic Value (IV)} = \sum_{i=1}^{\infty} \frac{CF_1}{(1+r)^1} + \frac{CF_2}{(1+r)^2} + \frac{CF_3}{(1+r)^3} + \cdots + \frac{CF_i}{(1+r)^i} \tag{2.3.1}$$

Whereby i is a time period and CF is the cashflow. Therefore, CF_1 is the cashflow from the period 1 (i.e. year 1). Moreover, r is the discount rate introduced by Williams. Although this idea is far away from simple, it is now popular among investment people. A detailed view and explanation of the idea can be viewed in [93].

Besides the discounting strategy, the intrinsic value is the indicator for the real value of an asset. Its market price rather can be higher or lower. If an investor is able to purchase an asset below its IV, he/she is likely to make profit [56, p.30].

2.3.2. Castle-in-the-Air Theory

Another theory to predict the future value of an asset is the *Castle-in-the-Air* concept. Instead of focussing on calculable facts it uses psychic values. In 1936 Keynes [50] introduced this method as an alternative method for professional investors to beat the market [56, p.31]. His approach is rather than looking directly at the future values direction to analyze the crowd of market participants and to anticipate how they will behave. The name of the theory originates from the strategy to estimate which investment options are most likely to gain attention from the crowd. Those market participants build in an optimistic period their hopes into castles in the air. A psychological investor, therefore, should buy those assets beforehand to make a profit when the crowd purchases those [50, p.31].

Malkiel [56, p.32] emphasizes that it is not important within this concept which price you pay for an asset. The only critical criterion is that somebody is willing to pay more in the future. This is also known as the *greater fool* theory.

2.3.3. Candlestick analysis

Section 2.1 already describes the overall concept of candlesticks. This section presents the different methods how those can be visually interpreted to gain information about future directions. Therefore, the candlestick analysis is part of the *charting* methods [25, p.527].

The common 40 interpretations persist of one or a combination of more than one candles. As all other analysis methods for random walks, the candlestick analysis does not guarantee a successful prediction rather it shows a momentum of the current market status. The following three patterns give a brief over the different usages and consequential trading conclusions [37]:

Hammer / Hanging Man These patterns introduce a way to recognize possible turning points. A *hammer* is a candle with a small body, no upper shadow and a lower shadow which is at least twice as large than the body. Based on the position of the candlestick it is also called *hanging man* if it is recognized within a rising tendency instead of a falling one. Although both patterns are considered alone only as a small indicator for a turning point, those combined with the next candle shows a brief direction change. It abstracts a big fall followed by a great rise. The trading conclusion is, that if a hammer is recognized with a decreasing tendency the asset should be **buyed**. Otherwise, a hanging man indicates insecurity within the market during an uprising. Therefore, a **disposal** is recommended. The greater the difference between body and lower/upper shadow is, the more significant the corresponding pattern is.

Bullish- / Bearish Belt Hold The *belt hold* patterns consist of a big body and depending on its direction a smaller higher or lower shadow. The *bullish* version opens low but closes high and has a small high shadow. The *bearish* one is vice versa. The greater the body the greater is the impact of the pattern. A long shadow supports the prediction. A bearish belt hold indicates a further fall of the random walk, a bullish candle a further rise.

Bullish- / Bearish Engulfing Pattern The *engulfing* pattern contains two candlesticks. The first one is a white or black candle followed by greater inverse one. The only condition is, that the second candle needs to *surrond (encircle)* the primary body. The conclusion is again a turning point. For example, the *bullish* pattern is a black small candle followed by a great white one. In the context of a decreasing tendency beforehand, this predicts a rise of the asset and therefore a **buy** is recommended. The greater the difference between the candles the greater the implication of the event.

As already pointed out, there are over 40 patterns found in the classic Japanese candlestick analysis. More examples and explanations can be found in [58, pp.203-206].

2.4. Introdcution to neural networks

The idea of neural networks goes way back to the early research about the human brain. Approximately 10^{11} elements linked by roughly 10^4 relations per member form the basic structure of the man like intellect [41, Chapter 1 p.1, p.8]. Each unit of this biological nervous system is called *neuron*.

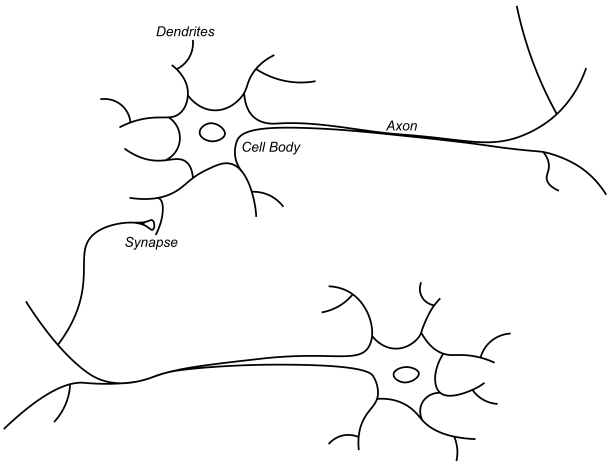


Figure 2.2.: Drawing of biological neuron

Figure 2.2 shows a simplified schematic diagram of a biological neuron with an input connection from another neuron. A unit is composed of three major segments: the dendrites, the cell body, and the axon [41, Chapter 1 p.8]. Signals within the system are carried through the dendrites which build a network of nerve fibers carrying electronic signals from other neurons to the cell body. The main element is the cell body which is processing the incoming signals and delivers the output via the axon to the rest of the network. Furthermore, the contact point between a cell body and a dendrite is defined as a synapse. While it is already explored that the synergy of the arrangement of the neurons and the strengths of the synapses deliver the highly effective value of the human brain [41, Chapter 1 p.8], there is only a small knowledge of the neuron's exact chemical way of functioning [51, p. 13].

Derived from the biological concept the basic *artificial* neural network idea emerged after a new approach to simplified electronics neurons by Warren McCulloch and Walter Pitts in 1943 [59]. The overall concept is also known as *connectionist models* or *parallel distributed processing* [51, p.13]. From there the evolution of neural networks undergone a long time of highs and lows. Main milestones are the discovery of *classical conditional* by Hebb [43] and Frank Rosenblatt's first application of a perceptron network with a corresponding learning rule in the early $1950s$ to achieve a limited pattern recognition [41, Chapter 1 p.3]. In 1969 Minsky and Papert [62], showed up the deficits of perceptron models by using mathematical analysis to show the exact limitations of a class of computing machines which are seen as serious models of the human brain. Although some researchers continued to search for new aspects, the overall studies were suspended [51, p.13]. In the $1980s$ the trend gathered pace again after a new wave of more powerful personal computers and the discovery of the error backpropagation algorithm [41, Chapter 1 p.3, 51, p.13]. The new algorithm was observed independently by several researchers but Rumelhart's and McClelland's [76] paper from 1986 was the particular answer to the criticisms of Minsky and Papert [41, Chapter 1 p.4].

The following section introduces the basic concept of artificial neural networks while explaining the architecture and different algorithms. Moreover, in sections 2.4.2 and 2.4.3 modern fields of application are described. The section closes with a selection of successful and unsuccessful implementations in different areas during the years.

2.4.1. Description

The structure of a neural network (NN) is based on a pool of simple processing units (PUs) connected through a large number of weigthed connections [94, p.391, 51, p.16]. This concept enables the NN to memorize experiential knowledge and process information without defining specific policies [94, p.391]. Therfore, it does not handle the data sequential but parallel and distributed [51, p.15]. Returning to the biological inspiration in the beginning of the section a neural network is based on the same fundamentals. The weight corresponds to the strength of the synapse while the activation function and the output represent the axon. The summation of the input with the weights and bias is comparable to the cell body. Rumelhart and McClelland [75] defined further

aspects [51, p.15], which are illustrated in fig. 2.3 and explained in more detail in the followed paragraphs:

- Every PU has an output state y_k which is derived from the activation function F_k.
- Every connection between processing units is determined as a weight w_{jk} which affects the impact of an element j on the unit k.
- Every item has an activation function F_k determining the new level of activation based on the quantified input $s_k(t)$ and the current activation $y_k(t)$.
- Every processing unit has an external *bias* (or *offset*) θ_k.
- Each NN has a learning rule collecting information.

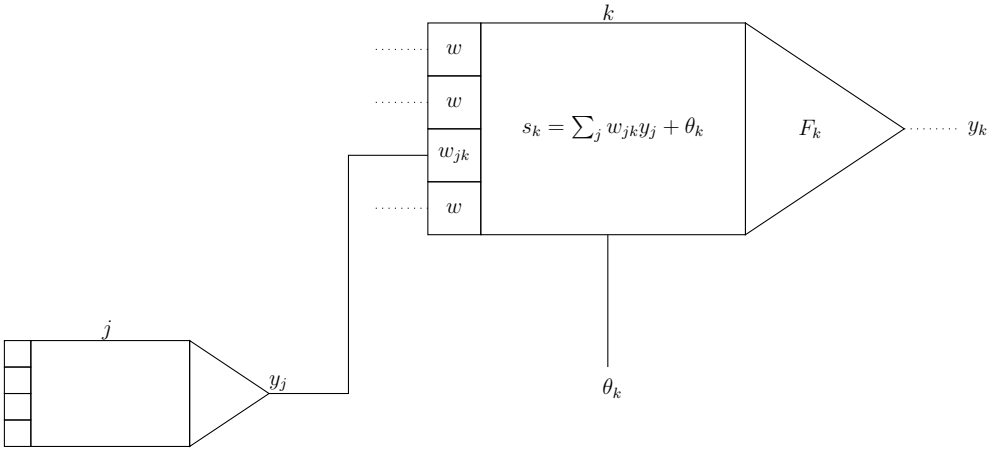


Figure 2.3.: Fundamental components of an artificial neural network with a weighted summation rule. [51]

Processing units

The fundamental and also smallest unit within a neural network is the processing unit. Thus the major function of a PU is to calculate an output value based on a computation of the weighted inputs in combination with a certain threshold. Thereby, the outcome y_k can be a sum of the weighted sums of the inputs y_j. The quantifier w_{jk} represents the weight of the connection between the output of a neuron j and the current processing

unit k. The quantifier w determines the impact of an input on the PU. This can be mathematically represented as [94, 51]:

$$\begin{aligned} y_k &= F_k(\sum_{j=1}^{n} w_{jk} y_j + \theta_k) \\ &= F_k(a_k) \end{aligned} \tag{2.4.1}$$

While the activation levels are representing the short-term memory of an artifical neural network, the whole weigth matrix can be considered as the long-time memory [94, pp.391f] since it reflects the skills that it has earned during the training process (see *Training*). The bias θ is a correction method similar to a constant within a linear function $y = ax + b$. It can be used to shift the entire transfer function to a specific amount to optimize the results. In general an offset is not required but networks which have one are considered as more powerful since those have an extra variable. This parameter can *e.g.* avoid that a network with an input vector of zeros always outputs a zero as well.

Connections

As already described a connection is defined as the passing on of the output from a processing unit to another one. Thereby, the weights of a contact represent the strength of it. If W_{jk} is positive it is considered as *excitation*. Otherwise, a negative weight as *inhibition* [51, p.16].

Each PU can be assigned to one of the following types depending on how they are interconnected through the network [51, p.16]:

Input The unit's input data is from outside of the network.

Hidden Input and output connections from the neuron are within the network.

Output The element receives input data from within the network but sends its result out of the NN.

Since a neural network works as a parallel distributed system multiple processing units can be calculated at the same time. More precisely this can be achieved via a *synchronously* or an *asynchronously* approach [51]. A synchronous calculation process is

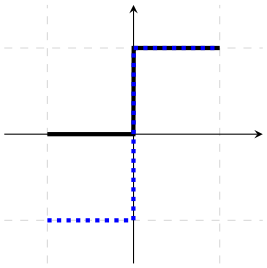
a concurrent method. That implies that all neurons update at the same time. With an asynchronously system, each PU update its activation at a specific time t.

Activation functions

The activation function (also known as *transfer function*) is the rule which determines the output of the neuron based on the total input and the current activation level [51, p.16]. Therfore eq. (2.4.1) needs to be supplement.

$$y_k(t+1) = F_k(y_k(t), a_k(t)) \tag{2.4.2}$$

Such an F_k can be linear or nonlinear of n [41, Chapter 2 p.3] but it is often implemented as threshhold function [51, p.17]. This threshhold option is not depending whether the function is linear or nonlinear but it is limiting it to a certain minimum and maximum value. One important performance factor is the transfer function. Thence, for every problem a new evaluation for an activitation function needs to be done. The following figures show some of the common transfer functions [41, Chapter 2 p.6, 51, p.17, 94, p.392].

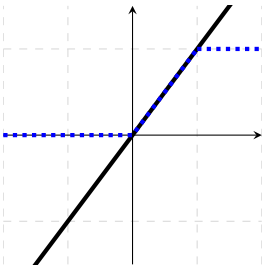


$$F_k(a_k) = \begin{cases} 0, & a_k < 0 \\ 1, & a_k \geq 0 \end{cases} \tag{2.4.3}$$

$$F_k(a_k) = \begin{cases} -1, & a_k < 0 \\ 1, & a_k \geq 0 \end{cases} \tag{2.4.4}$$

Figure 2.4.: Stepfunction *(dotted: Symmetrical Stepfunction)*

The *hard limit* activation function (*or: stepfunction*) is used to determine a binary classifciation problem. Equation (2.4.3) shows that the output of the neuron is 0 when the summarized inputs are below 0. Otherwise, it is 1. A modified version of this function is Equation (2.4.4) whereby the dotted graph is symmetrical to the x axis. The result for below 0 is hereby -1 [41, Chapter 2 p.4]. Both functions are visualized in fig. 2.4.

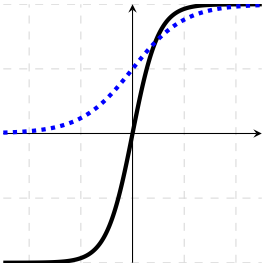


$$F_k(a_k) = x \tag{2.4.5}$$

$$F_k(a_k) = \begin{cases} 0, & a_k < 0 \\ x, & 0 \geq a_k \leq 1 \\ 1, & a_k > 1 \end{cases} \tag{2.4.6}$$

Figure 2.5.: Linear *(dotted: Saturated linear)*

Figure 2.5 anticipate the *linear* and *saturated linear* transfer function. Thereby, eqs. (2.4.5) and (2.4.6) show the logic behind the graphs. The linear function does not have any limits. The saturated linear activation function limits it to 0 to 1. [41, Chapter 2 p.4]



$$F_k(a_k) = \frac{e^{a_k} - e^{-a_k}}{e^{a_k} + e^{-a_k}} \tag{2.4.7}$$

$$F_k(a_k) = \frac{1}{1 + e^{-a_k}} \tag{2.4.8}$$

Figure 2.6.: Hyperbolic Tangent Sigmoid *(dotted: Log-Sigmoid)*

The *hyperbolic tangent sigmoid* and *log-sigmoid* are shown in fig. 2.6. Equation (2.4.7) is the equivalant to the $\tanh(A_k)$. For multilayer networks, this function is often squashed to a threshhold function shown in eq. (2.4.8) as it is differentiable. The log-sigmoid activation function limits it to 0 to 1 (see eq. (2.4.8)). [41, Chapter 2 p.5]

Training

A neural network's goal is to produce an output or a set of outcomes which solves a given problem. Therefore it changes its weights of each processing unit to build some kind of memory and intelligence [51, p.18]. This is the same approach currently known about the human brain: Some brain structure is developed during pregnancy based on the genetic material of the parents. Thus, the most parts are evolved through learning by creating new connections, destroying old ones or adjusting strengths of synapses [41,

Chapter 1 pp.8f]. In the area of artificial neural networking this is known as *priori* knowledge (already given information results) and *training* [51, p.18]. Therefore, one way is that the NN processes training data which is a set of input data and a given correct output (see *supervised learning* later). How the interconnected system performs its weight adjusting is depending on the problem and given data. Mathematically expressed this is [94, p.393]:

$$\frac{dw_{jk}}{dt} = f_L(L, w_{jk}, t) \tag{2.4.9}$$

Whereby t is the time and L is the weigth adjustment information. f is the choosen learning function. Equation (2.4.9) is performing the weight changes. The changes to the corresponding connection jk are saved in L. The learning function f_L uses this information to modify w_{jk}. After this is applied to every neuron of the NN the current learning round is finished for a training sample.

Usually the learning strategies are categorized in two distinct sorts [51, p.18, 94, p.393]:

Supervised learning *or Associative learning* This learning concept is based on an external guidance. Hence, it uses input pattern (*local information*) in combination with the matching output pattern (*external information*) to train the neural network. The desired solutions (outputs) are usually provided by an external source. If they are supplied by the system which contains the model it is called *self-supervised*.

Unsupervised learning *or Self-organisation* In comparison to the supervised learning pattern, the unsupervised one only depends on local information. Therefore, the NN needs to discover statistically salient features to build a correct solution on its own. More information on competitive learning algorithms can be found in [4].

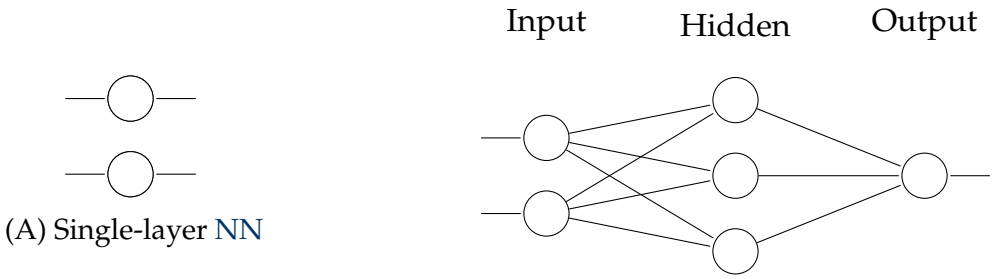
Examples of both learning strategies will be explained in section 2.4.3.

2.4.2. Types

Each problem requires a specific neural network solution. Therefore, over the years different network architectures have been developed and published. All of them share the same basic of processing units and connections, but the arrangement and interconnections differ [94, p.393].

First, NN can differ in the amount of layers. A layer is a pool of PUs which operate in parallel [41, Chapter 2 p.9]. The simplest structure is a *single-layer* architecture which is composed of only n neurons. Those are connected to every input and their result is the output of the network.

The next step is to add more layers. If there is a total of two or more layers the network is considered as *multi-layer*. Since the first layer is the *input layer* and the last one is the *output layer*, the layer in between are distinguished as *hidden layers* [41, Chapter 2 pp.9ff]. Those are not visible for the environment [94, p.393].



(A) Single-layer NN

(B) Multi-layer NN

Figure 2.7.: Schemas of single- and multi-layer NNs

Another architecture decision is the type of the NN. A common structure is the *feedforward* type. It does not have any feedback loops and the information flow is relatively simple like seen in fig. 2.7B. The data is inserted in the PUs in the input layer, processed and transferred to the hidden layer and finally emitted after the output layer (see section 2.4.1).

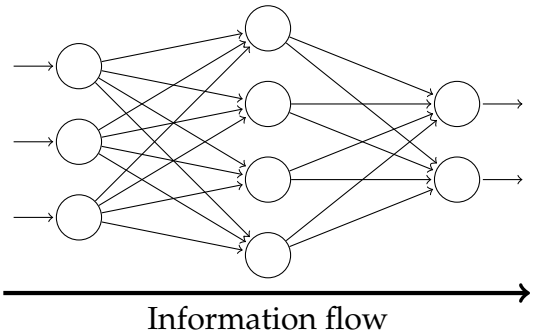


Figure 2.8.: Schema of a multi-layer feedforward NN

In contrast to the feedforward neural network,the *recurrent* (or *feedback*) architecture does not have a single-way data flow. The connections in this type are bi-directional,

which means that the output is not only redirected to next layer of neurons but to the neuron itsself [41, Chapter 2 p.14, 94, pp.395f]. Often this is implemented as a delay module at the output of the neuron. This is described in detail with the aid of an algorithm in section 2.4.3.

Since there are a lot of different combination possible, there are some common *best practices* for the architecture of neural networks:

- The number of inputs to a layer should be different from the number of the PUs in the layer [41, Chapter 2 p.9].
- It is not required that every neuron within a network should have the same activation function [41, Chapter 2 p.10].
- Three step program to pick an architecture based on the problem specification by Hagan et al. [41, Chapter 2 p.19]:
  1. "Number of network inputs = number of problem inputs
  2. Number of neurons in output layer = number of problems output
  3. Output layer transfer function choice at least partly determined by problem specification of the outputs" Hagan et al. [41, Chapter 2 p.19]

2.4.3. Current methods

Error measurement

One of the most important features of a neural network is its training ability. To measure how good the current configuration is, some fault indicators are necessary. Kröse and

Smagt [51] propose therefore based on Rumelhart, Hinton, and Williams [76, p.534] the following equations:

$$E^{y_j} = \frac{1}{2}\sum_{k=1}^{N_k}(d_k^{y_j} - y_k^{y_j})^2 \tag{2.4.10}$$

$$E_{learning} = \frac{1}{P_{learning}}\sum_{y_j=1}^{P_{learning}} E^{y_j} \tag{2.4.11}$$

$$E_{test} = \frac{1}{P_{test}}\sum_{y_j=1}^{P_{test}} E^{y_j} \tag{2.4.12}$$

$$E = \frac{1}{2}\sum_{y_i=1}\sum_{k=1}(y_k^{y_j} - d_k^{y_j})^2 \tag{2.4.13}$$

Equation (2.4.10) is the total quadratic error for the difference between output of k and the desired output while training the network with learning data. Hereby, $d_k^{y_j}$ is desired output for the processing unit k when input pattern y_j is used [76, p.534]. The *learning error rate error* is displayed in eq. (2.4.11). This calculates the average error per learning sample represented in eq. (2.4.10). Furthermore, eq. (2.4.13) defines the total error E of a network [76, p.534]. Kröse and Smagt [51] propose to define the *test error* rate as the average error of the test set (see eq. (2.4.12)).

Perceptron network

As in the history of neural networks (see section 2.4) already mentioned Frank Rosenblatt's perceptron network was one of the milestones in the evolvement of NNs. This paragraph is a simplified single-layer pattern recognition example of his original implementation. Therefore, there is a training set composed of an input vector y_j and an output vector $d(y_k)$. Another simplification is that the example assumes the result differs only between -1 or $+1$ [51, p.24] which ranks it as a binary classification problem.

Rosenblatt's learning rule consists of an easy supervised algorithm shown in fig. 2.9. The weights are initialized with random numbers. Afterward, the algorithm iterates through every data vector of the training set and corrects the corresponding weights, if

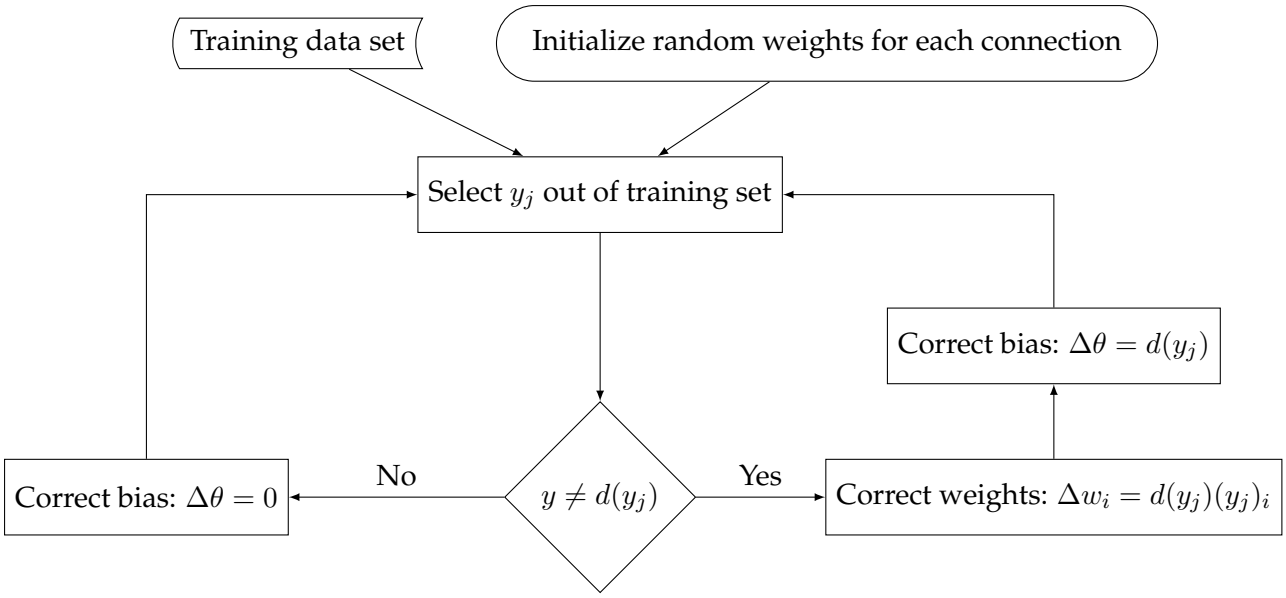


Figure 2.9.: Flowchart of the perceptron learning algorithm [51, pp.24f]

the result y_k is not equals to the desired one $d(y_k)$. Furthermore, the bias is corrected according to this function [51, p.25]:

$$\Delta\theta = \begin{cases} 0 & \textit{if the perceptron responses correctly} \\ d(y_k) & \textit{otherwise} \end{cases}$$

One extended aspect of the perceptron concept is the convergence theorem. For a detailed review on this theorem see Kröse and Smagt [51, Chapter 3.2.2].

Delta rule

Another basic supervised training algorithm is the *delta rule* initialy proposed by Aleksander and Morton [2]. The method tries like all other learning algorithms to optimise the output y_k by finding such an weight matrix w that the output is as similar as possible to the desired output: $y_k \approx d(y_k)$ [22, p.287].

$$(dw)_{ji} = n(y_j - (wx)_j)x_i \tag{2.4.14}$$

In the context of the delta rule eq. (2.4.9) is modified to eq. (2.4.14) [75, p.332], whereby, n is a chosen parameter called the *learning rate*. Furthermore, in this context x_i is a

learning sample at position i in the training set (x, y) with the desired output y_i. $(wx)_j$ denotes the jth element of wx.

The delta rule is, in the end, just a method to optimise the weights step by step to the ideal weight vector. It has been shown, that this rule works best with models without any hidden layers. Otherwise, NN with hidden layers are not suitable for it. [77]

Back-propagation

The back-propagation is also known as the *generalized delta rule* whereby the objective is to "find a set of weights that ensure that for each input vector the output vector produced by the network is the same as (or sufficiently close to) the desired output vector" [76, p.534]. The overall architecture is originally a multi-layer type. However, particularly there are no connections within a layer or between layers in the direction from the output to the input neurons allowed. Though an interconnection can skip hidden layers [76, p.533]. The total input x_k of a PU the sum of the weighted inputs $y_j w_{jk}$ [76, pp.533f]:

$$x_k = \sum_j y_j w_{jk} \tag{2.4.15}$$

$$y_k = \frac{1}{1 + \exp(-x_k)} \tag{2.4.16}$$

Equation (2.4.16) is its real-valued output. Equations (2.4.15) and (2.4.16) are proposed by Rumelhart, Hinton, and Williams [76] but those do not need be mandatory exactly the same [76, p.534].

To train the network the algorithm uses two phases:

First phase: *Feedforward* An input x out of the training set is processed through the network similar to a simple feedforward NN. Thereby, the outputs of all neurons are calculated [76, p.534, 51, p.36]. Furthermore, each result is compared to the desired result which gets documented in the error E^{y_j} (see eq. (2.4.10)).

Second phase: *Backward* The generated error signal is transferred from the output layer backward to the input layer. Each element calculates its own appropriate weight adjustments [76, pp.534f].

2.4.4. Performance indicators by Kröse and Smagt

Section 2.4.2 already introduced various aspects to specify a neural network architecture but which factors needs to be considered to reach the optimal performance?
Kröse and Smagt [51, Chapter 4.8] introduces a study about the impact of different variables on the performance and approximation error. Firstly, the book defines that the learning algorithm and number of iterations of the learning set determines how optimized the error on the training set is. Furthermore, the amount of learning data resolves the quality of the NN. *Quality* is in this context how "good the training samples represent the actual function" [51, p.43]. Finally, the quantity of hidden neurons influences the "expressive power" [51, p.43] of the system.

Equations (2.4.10) to (2.4.12) are the used measurements methods to evaluate the errors looking at the learning set size and number of hidden units [51, pp.43f]:

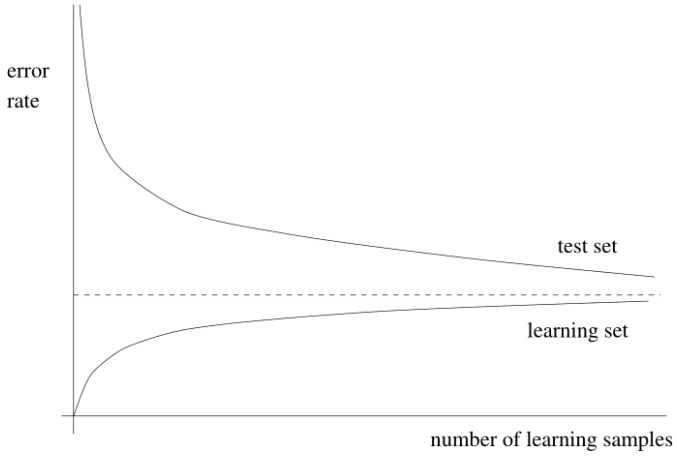


Figure 2.10.: "Effect of the learning set size on the error rate. The average error rate and the average test error rate as a function of the number of learning samples." [51, Figure 4.8]

Number of learning samples Kröse and Smagt's experiment is based on the approximation of the function $y = f(x)$. The test compares various sizes of the learning set to contrast them to subsequent errors $E_{learning}$ and E_{test}. Figure 2.10 shows the results of the different set sizes and compares the error rate of the test set and learning set. As seen the error rate of the training samples rise with the number of samples. This is due that the NN needs to fulfill more possible correct results. In the meantime, the test set error decreases over time. In more detail, they compare the amount 4 and 20, which graphs can be seen in fig. A.1. In the first

case, $E_{learning}$ is small but E_{test} was large. With 20 learning data samples, the effect is quite different. E_{test} is quite low but $E_{learning}$ is higher. The authors conclude that a modest learning rate on a small learning set is not a reliable performance indicator.

Number of hidden processing units Kröse and Smagt use the same experimental setup but now vary the amount of hidden neurons. During the experiment they created the so-called *overtraining* (or *overfitting*) effect. Due to a large number of PUs within the hidden layers, the network is fitting the noise of the learning data instead of smoothing the approximation. Therefore, the authors infer that although a high number of hidden neurons lead to a small error on the learning set, the overall network is not skilled in working with new data. Their results can be seen in section A.1.

2.5. Current state of art

Sitte and Sitte [78] try to predict the S&P 500 financial from 1973 to 1994 time series by the sum of their long-term trend f and a residual series r. Whereby, r is the detrended (by subtracting a least mean square fitted exponential function) S&P 500 data. They modified the data since "NNs have difficulty to predict strongly growing time series" [78, p.166] like the original series. Furthermore, Sitte and Sitte [78] normalize and rescale r to zero mean in the range of $[-0.8, 0.8]$. This should avoid the saturation regions of the sigmoid transfer function. The authors compare two network types. The first one, a Time Delay Neural Network, is a feed-forward network with one hidden layer which has an experiment range between 2 and 32 PUs. Furthermore, the amount of the historical data, called *window size*, was adapted from one day up to one month. They use different activation functions in the hidden layer (hyperbolic tangent) and the output layer (linear). The second network is an *Elman recurrent* one, whereby the context units are progressively increased from 2 to 16. Both networks' training algorithm is *Levenberg-Marquardt* and are implemented in the Matlab Neural Network Toolbox. The data is split in 60% learning and 40% testing data.

In the end, both networks output almost identical results. The authors conclude that there are "simply no more information the detrended S&P 500 time series" [78, p.168] which contradicts the theory that historical data of time series can be used to predict future ones. Sitte and Sitte [78] explain their results and test further factors in their article "Neural Networks Approach to the Random Walk Dilemma of Financial Time Series".

Adhikari and Agrawal [1] try to forecast financial time series by combining feedforward ANNs (FANN), the random walk model and Elaman artificial neural networks (EANN). The hybrid's development is based on the problem of time series composed out of linear and nonlinear components (see eq. (2.5.1)). Therefore, the authors separate the original TS (Y) into a linear (X) and nonlinear (Z) part by estimating the linear one with a random walk model and subtracting this from the original series (see eq. (2.5.2)). E is the resulting rest including the nonlinear part.

$$Y = X + Z \tag{2.5.1}$$

$$E = Y - X_{estimated} \tag{2.5.2}$$

$$Z_{estimated} = \frac{1}{2}(E^{FANN}_{estimated} + E^{EANN}_{estimated}) \tag{2.5.3}$$

$$Y_{estimated} = X_{estimated} + Z_{estimated} \tag{2.5.4}$$

In eq. (2.5.3) $Z_{estimated}$ is the result of average the predicted values of a FANN and an EANN estimation. The auhtors discuss the sailent features in "A combination of artificial neural network and random walk models for financial time series forecasting" [1, p.1445]. After appyling the concept to four real-world financial time series, the results (eq. (2.5.4)) can be seen in fig. 2.11, whereby, the dotted lines are the predictions and the normal ones are the actual time series. The authors conclude that their hybrid method adds a "substantially improved the overall forecasting accuracies and also outperformed each of the individual component models" [1, p.1448].

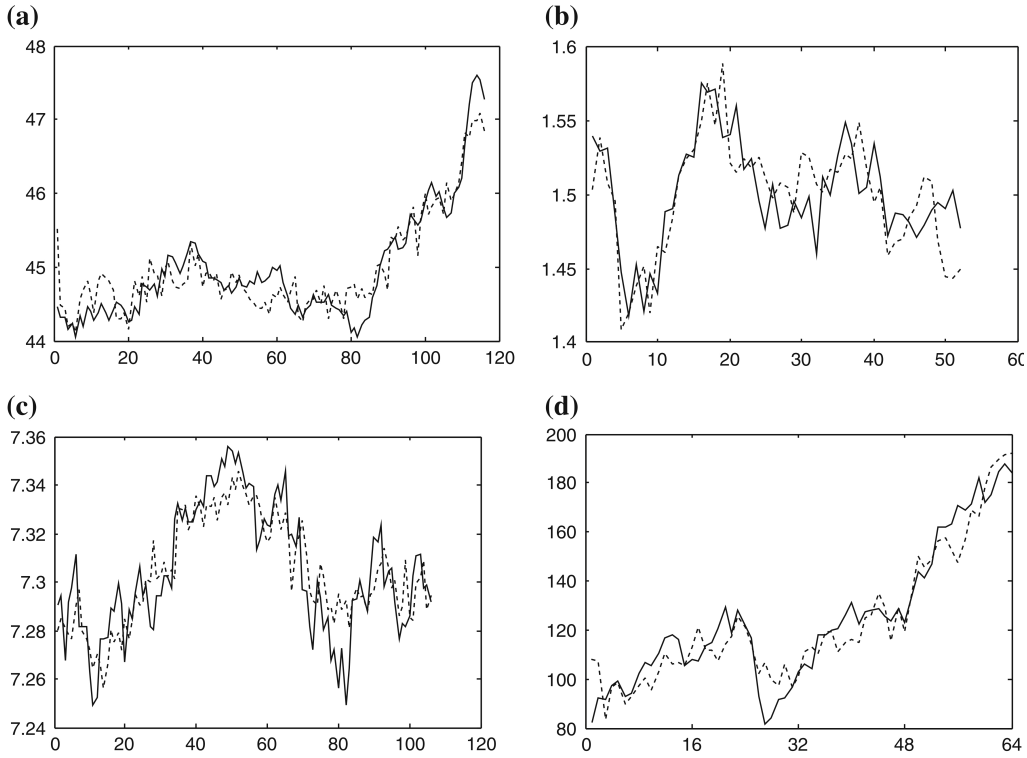


Figure 2.11.: Graphs of actual and predicted datasets through the hybrid method for **a)** USD–INR exchange rate, **b)** GBP–USD exchange rate, **c)** S&P 500 index, **d)** IBM stock price [1, Fig. 5]

Furthermore, there are various publications about neural networks for financial time series but due to their commercial value, it is highly unlikely that successful prediction designs are made public.

3. Practical evaluation

This chapter starts with an describtion of the scoring criterias which will be used in section 3.3 to evaluate the in section 3.2 described frameworks. The final decision with a detailied explanation is closing this chapter.

3.1. Evaluation criteria

The first considered aspect is the framework's *environment*. Which coding language is used and how old is the latest version, are measurements to score the structure. This includes the regularity of releases and the overall repository maintenance. Moreover, if the documentation is comprehensive and easy to understand, the score in this criterium is higher.

Since the results of this thesis should be used in a professional environment, it is important to use a long-term, scientifically proved framework. The criterium *confidence* considers a good structure to be under the patronage of a admired company, an internationally respected university or based on an active open-source community with more than 1000 participants. In all cases, the project should be financed for the next few years. The internal calculations of each framework are rated on the present documentation, their developers, and publications. Another observable factor in this criterium is the overall usage of the code. If it is adopted by big companies and has a large community with help forums, tutorials or regular mailing lists, it is scored higher.

The third criterium is the the *quality assurance* (*short: quality*) of the package. Is there a reasonable test coverage and on which concept are those based. Furthermore, it is

important whether the developers follow consistently specific guidelines to maintain a good code quality.

In addition to the level of confidence, for professional usage the *license* is major fact as well. A free, open framework is favored.

The fifth criterium are the *technical environment* (*short: technic*) which considers the current state of the framework. If the technical dependencies in hard- or software are low the score is better. This includes hardware, operating system or other framework dependencies. Moreover, the ongoing amount of known bugs is also covered. Furthermore, it is scored whether the codes main goals are consistent. One of the most important aspects is the amount of major *features*. The framework is considered as good when all necessary features for this thesis are covered. This includes different activation functions (the more the better), various methods to architect a network (the more the better) and a reasonable amount of functions to read data. Thus, the score is lowered when there are overweighted functions not useful for this project.

Another important aspect is *performance*. This criterium considers a good framework to have a fast response time. The initialization, training and testing performance is observed as well, but since these backtests are not real-time dependent, those are only a small factor. Thus, real-time learning to improve the neural network with new information is a strong scoring aspect. Performance boosters like GPU support or special hardware lead to a higher score, as long as this does not interfere with a normal use or sets any limitations.

3.2. Frameworks

In the following, there is a quick overview of various frameworks with short descriptions (in alphabetical order).

Caffe is a deep learning framework developed by Berkeley Artificial Intelligence Research. It is based on a Ph.D. project from Yangqing Jia and currently under the leadership of Evan Shelhamer, a current Ph.D. student at UC Berkely. [49]

Caffe2 is a successor from Caffe developed by Facebook who recently made it available to the public (open source). The lead developer is Yangqing Jia which started the first version [57]. The improvements to Caffe are better hardware support, greater scalability and a higher stress test. [79]

Encog is a machine learning framework for Java and C#. The developer Jeff Heaton (Ph.D.) is a data scientist and an instructor at Wahington University. The project is focused on neural network algorithms. [42]

Keras is a high-level neural networks API for *e.g.* Python, Theano or TensorFlow. It was initially developed by the Open-ended Neuro-Electronic Intelligent Robot Operating System project but is currently maintained by its main developer François Chollet. [14]

Lasagne is similar to Keras an extra library which is specialized on building and training neural networks in Theano. The core eight-man team around Sander Dieleman developed the framework from 2014 to 2015. Currently, it is an open source project with 62 contributors. [19]

Scikit-Learn is a framework designed for data-mining and data-analysis powered by Google, Columbia University, and different other institutions. After the Google Summer of Code, the project was started in 2007, in 2010 the first release was published. The framework is built for and on Python. [68]

TensorFlow is a package based on multidimensional data arrays called *tensors*. Since 2015 the Google Brain Team is developing it to calculate conducting machine learning problems and deep neural networks. Currently, it is used by various companies including Google's own products. [85]

Theano is a Python defined language to represent and manipulate mathematical expression. Although it has an active developer community (currently over 250) it is mainly developed by a team from the Université de Montréal. [89]

The complete evaluation scheme is discussed in section 3.3 but before describing the frameworks in more detail, *Encog* and *Lasagne* are removed from the competition. Both packages do not have the essential patronage behind their projects. Therefore, it is not guaranteed, that every calculation and step is correctly implemented. Moreover, it is not assured, that the developers continue to maintain their code. Whether *Keras* fulfill these indispensable requirements is discussed later.

3.3. Evaluation

In this section the frameworks get ranked based on the in section 3.1 discussed criteria. The scores' values are between 0 and 10, whereby, a zero in one category or sub criteria is the direct disqualification for a framework. The final result is a weighted sum of all seven criteria by which 100 is the maximum value. Table 3.2 shows the final marks in each category. Moreover, each of those has different sub criteria whose impact on its main criteria is shown the table.

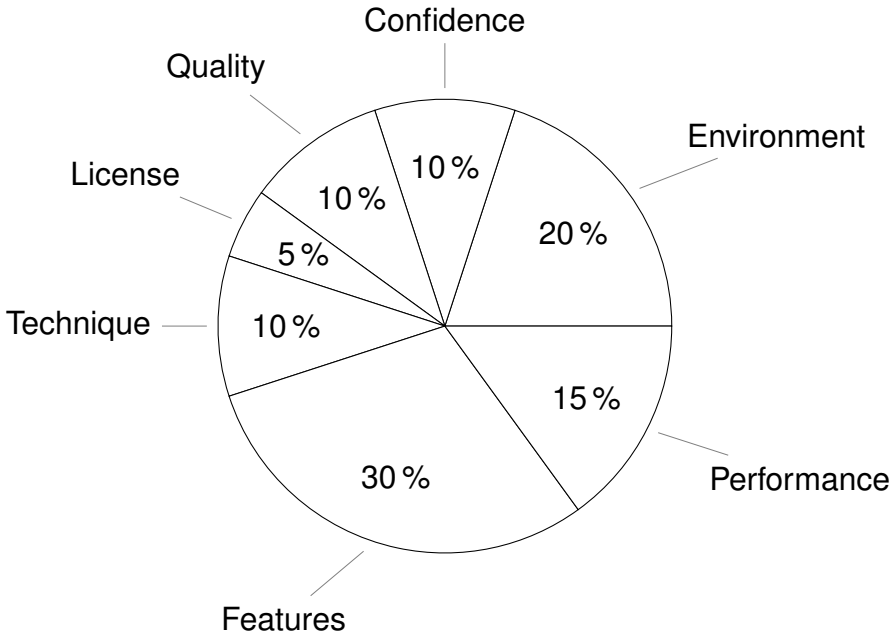


Figure 3.1.: Procentual weighting of the evaluation criteria

The weight of each category is shown in fig. 3.1. The evaluation criteria and their weights are chosen based on the scope of this thesis. As seen in table 3.2, Keras is disqualified as it is only a high-level implementation of other frameworks covered in the evaluation. In combination with the low level of confidence as there is only one main developer, it was removed from the final decision. The scoring may vary depending on the use case of the evaluation.

		Caffe	Caffe2	Keras	Scikit-Learn	Tensor-Flow	Theano
Language	10%	*7.5*	*7.5*	*0*	*6*	*9*	*6*
Releases	40%	*5*	*9*	*9*	*5*	*9.5*	*7*
Documentation	50%	*9.5*	*9.5*	*8*	*9*	*8.5*	*7*
Environment		**7.5**	**9.1**	**7.6**	**7.1**	**8.95**	**6.9**
Patronage	50%	*6*	*9*	*1*	*8*	*10*	*6.5*
Future	20%	*6*	*8*	*3*	*9*	*9*	*6*
Usage	30%	*9.5*	*8*	*3*	*10*	*10*	*6.75*
Confidence		**7.05**	**8.5**	**2**	**8.8**	**9.8**	**6.475**
Test coverage	40%	*10*	*5*	*5*	*10*	*8*	*5*
Guidelines	60%	*10*	*8*	*9*	*9*	*9*	*5*
Quality		**10**	**6.8**	**7.4**	**9.4**	**8.6**	**5**
License		**9**	**9**	**9.5**	**9**	**9**	**9**
Dependencies	50%	*8*	*9*	*5*	*7.5*	*9.5*	*8*
Known bugs	30%	*7.5*	*6*	*1*	*8*	*8.5*	*8*
Code goals	20%	*8*	*8*	*7*	*9*	*8*	*8*
Technique		**7.85**	**7.9**	**4.2**	**7.95**	**8.9**	**8**
Activations ftk.	35%	*8*	*7.5*	*7*	*9.5*	*8*	*7*
Neural networks	60%	*7*	*8.5*	*5*	*9.5*	*8*	*7*
Rest	5%	*10*	*5*	*9*	*8*	*7.5*	*6.5*
Features		**7.5**	**7.975**	**5.9**	**9.425**	**7.975**	**6.975**
Creation	15%	*9*	*7*	*7*	*8*	*9*	*8*
Real-time	60%	*9*	*8*	*7*	*9*	*8*	*7.5*
Booster	25%	*10*	*8*	*7*	*2*	*9.25*	*5*
Performance		**9.25**	**7.85**	**7**	**7.1**	**8.4625**	**6.95**
Final Score		**80.76**	**81.6**	**Disq.**	**83.775**	**86.32**	**69.13**

Table 3.2.: Scoring of the evaluated frameworks

This method also enables to build kiviat diagrams which visualize the strengths and weaknesses of each package. The figures for the evaluated packages can be viewed in figs. A.5 to A.10.

3.4. Decision

The evaluation revealed the strengths and weaknesses of each framework. Nevertheless, four packages have good scores in the mid-eighties. Caffe and Caffe2 with a total score of 80.76 and 81.6 are usable frameworks. Thus, it is not assured that Caffe will be expanded and maintained in the future. With Caffe2, Facebook launched a strong competitor and with features like simple conversion from Caffe networks to Caffe2, Facebook tries to actively head-hunt Caffe users. Furthermore, they already head-hunt the originally main developer Yangqing Jia to create confidence in their product. Although Caffe2 is actively and continuously developed, the framework was just released (July 2017). Therefore, there are not many reviews and reports about the implementation. It is also not known, whether the company will assure a long-time development since it just started. If it fails, it could be taken off the market in a few weeks.

The second rank in the evaluation is Scikit-Learn. Although it is used in a wide range of application, the focus is not very specific. The developers themselves say that "is designed to be easy to install on a wide variety of platforms" [17]. This manifests in their decision to not integrate GPU support and reinforcement learning in the near future. Although it can be implemented with the help of other packages, this leads to a high score but not the ideal framework for this thesis.

The framework which will be used is TensorFlow which is founded by the evaluation's final score of 86.32. It will be further described in chapter 4.

4. Implementation

The idea of the implementation is to build a neural network prototype that receives input of historical data which is already saved within the time series. This concept is derived from the *efficient-market hypothesis* by Fama which is described in section 2.1.

4.1. Architecture

Firstly, in this prototype, there is no database included. The initial idea to use the IBM Bluemix platform miscarried as it does not support the chosen framework TensorFlow. A workaround which involves docker would have exceeded the scope of this thesis. The current prototype runs locally in a Python environment.

The final decision to use TensorFlow involves the decision for the coding environment. The framework is build for and on *Python* (and *C++*). Although, there are libraries for *Java*, *Go* and *C*, those are currently not covered by the API stability guarantees from TensorFlow [87].

Python is an interactive, interpreted, and object-oriented programming language. A major difference to traditional codebases is that Python is syntax sensitive. This indicates, that different code layers are differentiated with indents. [32] The programming language is available in two versions: $2.X$ and $3.X$. The backstory is that in the early 21st century the developers decided to change some core features. Although older Python based programs can be converted to a $3.X$ version with the *2to3* tool, some modifications cannot be done automatically. Since the developers do not want to abandon users who have not done the transition yet, a two versions codebase is currently running simultaneously. Since in the future Python 3 should be the only standard, the

prototype uses Python 3.6. [10, p.17] One major advantage of Python is its wide variety of modules. Especially, a lot of deep learning software is only available in Python. More information about Python can be found on the official website [33] and different literature [10, 74, 54].

TensorFlow is the second generation of a machine learning systems developed by Google scientists and engineers. After creating and actively using the first generation *DistBelief*, Google simplified and rebuilt the code base to create TensorFlow. The first release was in November 2015 under the Apache license. The open source code currently can run only on single machines, although Google itself uses a distributed version within company products. [20, p.195] The code was developed for deep learning but the system's structure can also be used for other machine learning applications. To achieve this, TensorFlow is using a graph based structure whereby mathematical operations are represented as nodes. Connections between those are multidimensional data arrays called *tensors*. A more detailed description and explanation of the structure can be found in the official documentation [88] and different literature [20, 24, 97].

Another used framework is *Pandas* which is a Python module to create, handle and manipulate high-performance data structures. Although the project is community-driven, it is sponsored by the *NumFocus* group. Therefore, a commercial usage is possible. Furthermore, the usage is not application critical and can be easily switched to another framework. [69]

The whole predictor application is configurable with *JSON* files since in the end it should run automatically different networks with different structures. The major configuration is taking place in the *stockcodes.json* file which is build like shown in listing B.1. It defines the most important information for the neural network and used data:

name which specifies the name of the time series and therefore all its raw data input.

trainsize / testsize configures which day-based data should be trained and which one should test the neural network.

runsize is the amount of the data which is used to evaluate the NN.

nnconfig saves the parameters for the building of the network which is further examined in section 4.3.

forecast_sec defines the future time window which should be predicted.

window is the number of days which should be used to train the network starting from the last available dataset back counting. If it is set to 0 the whole data set is used.

parseDate has the value true if the timestamp input should be defined as six single inputs (year, month, day, hours, minutes, seconds) and set to false if it should be only a single input with the timestamp in ms.

predictPercChange changes whether the desired prediction should be the anticipated price or the percental change in comparison to the current price.

savedModelPath saves the path for the final model.

4.2. Data preperation

To build a neural network there has to be a large data set. This thesis raw data is always saved in one large file (*csv*) which contains tick based data of a time series. Tick based data means, that every time a change within the system happened (price, sell/buy, ...), a new dataset is generated. The amount can vary from only a few in several minutes to various in only a few seconds. In the following sections the used example data is the *YM* (Mini Dow Jones Indus.) series from *September 27th 2009, 18:00:00* to *November 30th 2016 18:59:20*. In more detail, this is tick based data. Therefore, sometimes there are multiple datasets per seconds. Sometimes there is for thirty seconds no new value. The total amount of information rows is $175.367.459$ and total file size is 5.1GB. An example line of data looks like:

09/27/2009,18:00:00,9622,1

Whereby, 9622 is the price of the series and 1 is the trading volume at this tick.

Before working with this data, it needs to be split, sorted and aggregated. This is done in two steps which are now further described.

DataParser

For parsing the huge data file this step is separating it into day-based single files. This can be initialized by passing an array of the codes to the function. Afterwards, it iterates over the name and preparing the following data structure:

1. Create a *status.json* file if none was detected in advance.
2. Check whether a local *data/data-name* folder exists, and create one when there is none. Each spurce file has its own folder titled after its name.
3. Check whether there already is a *status.json* file for the raw data. This JSON is saving all important information for the data transformation status. If there is no such file it creates one with the following structure:

```
{
  "status": false,
  "sorted": false,
  "data": 0,
  "from": 0,
  "to": 0,
  "sorteddata": 0
}
```

   status is storing if the file is already splitted and can be skipped for now. *data* is the total amount of valid data and *from* / *to* shows time lag of the raw data.
4. Calling the *seperateCSV* function to parse the file.
5. Saves new status in corresponding JSON file. For the mentioned example it looks like:

```
{
  "status": true,
  "sorted": false,
  "data": 175367459,
  "from": 1254074400000,
  "to": 1480532360000,
  "sorteddata": 0
}
```

The mentioned function in the fourth item is reading in the raw data file in 1GB chunks. Afterwards, it iterates over each row of the *csv* file and after a basic validation it parses the data and writes or appends it to a new *csv* file for the specific date. This is shown in listing B.2.

First, it parses the data's date in a Unix timestamp in milliseconds. Afterwards, it checks whether it is the oldest or newest data to write the time lag in the status file later. The next step is to determine the corresponding output file which is in this thesis always the day. In a lot of cases, tick based data is already sorted by time. To improve performance, it checks whether the last data set had the same output file. Therefore, it does not need to close and open a file for each iteration. In the last step, it is writing the data set with an internal unique id (*i*), the timestamp (*date_ms*) and its values to the corresponding file.

DataPreperation

Although separating the raw data file is already a kind of aggregation, the neural network cannot really work with this data yet. To prepare it, the following steps need to be taken:

1. Read in train-size, test-size, and run-size for each given time series. If these do not add up to 1, a warning is printed.
2. Call the preparation function for each configuration, which checks whether there is a status file and the parsing is finished.
3. Iterate through each day-based *csv* file and sort it (shown in listing B.3):
    a) Parse the data to a *Pandas Dataframe* which was already described in section 4.1.
    b) Aggregate the data to second-based data sets with the median price of each second and a total sum of all trades (volume).
    c) Sort the data based on their time stamp to achieve a chronological order of the day.
    d) Write the new data in an efficient *HDF5* file for better performance
    e) Return the total amount of aggregated data sets.

4. Return the sum of all aggregated data sets. This shows how much rows and data this procedure saves (represented by *sorteddata* in the status file).
5. Update the *status.json* file. For the mentioned example it now looks like this:

```
{
  "status": true,
  "sorted": true,
  "data": 175367459,
  "from": 1254074400000,
  "to": 1480532360000,
  "sorteddata": 31150557
}
```

6. Delete the old daily *csv* files. Those are not longer needed.
7. Create *sets* based ont the training-, test-, and runsizes. These sets of days and return an array with file names for each part.

Splitting, aggregating and sorting the raw data, in the example, saves $144.216.902$ data sets and a total of 4.3GB storage. Some test runs of the example lead to the following empirical benchmark results: the test machine was a Linux Ubuntu 16.10 with 30GB RAM, 50GB HDD storage, 8 CPUs and a NVIDIA M4000 graphic card with 8GB VRAM. TensorFlow is used in version $r1.3$ with GPU support. The *dataParser* took in average (5 runs) about ≈ 77 minutes. The *dataPreperation* took about 144 seconds.

4.3. Neural network

Like already pointed out in section 4.1 the whole application including the neural networks is configurable. Therefore, hidden layers, weights and biases are automatically created. Even train- and test-sets are generated in the runtime and cached on the local storage. This decreases the overall performance, but it is necessary to run automated benchmarks and neural network variants.

The general workflow is shown in fig. 4.1.

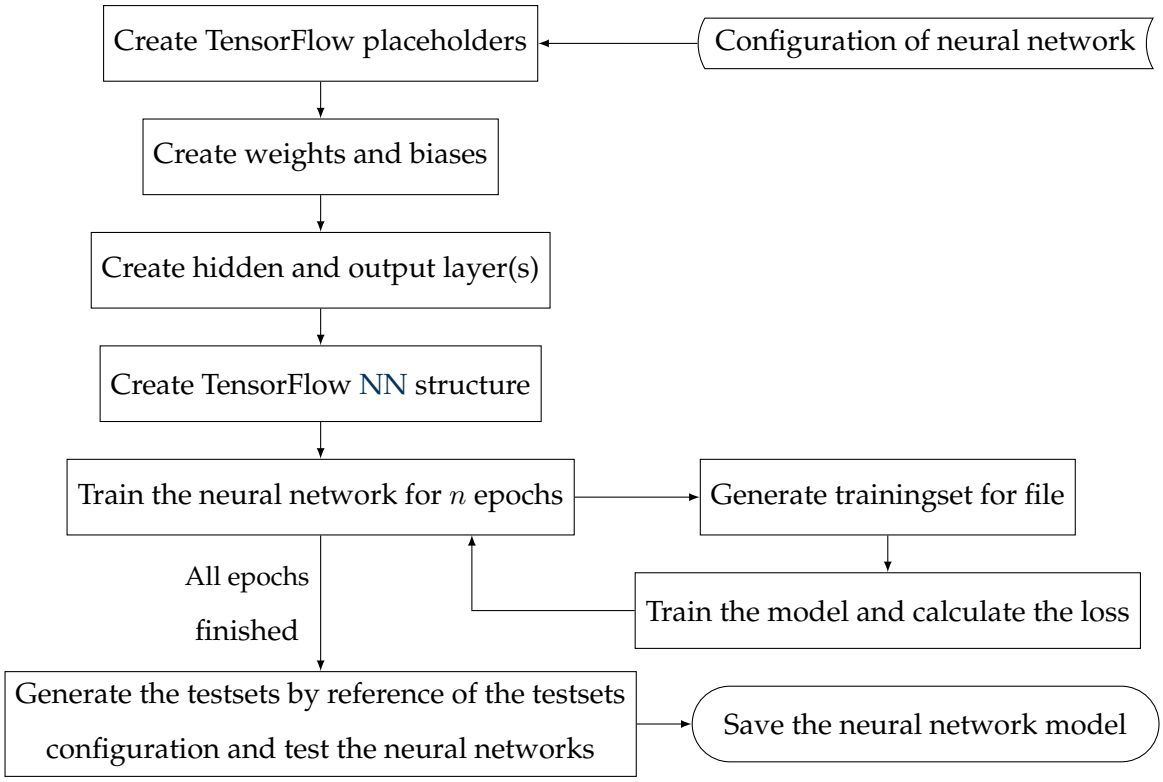


Figure 4.1.: Flow diagram for building and training of a neural network

The placeholders x and y are used to feed the NN with the training data. Whereby, x is the current dataset and y is the result. The *biases* are, when there are no predefined values in the configuration, generated with the function shown in listing 4.1.

```
def createBiases(config):
  biases = []
  for v in range(0, len(config["network"]["layer"])):
      biases.append(tf.Variable(tf.random_normal(
        [config["network"]["layer"][v]])))
  biases.append(tf.Variable(tf.random_normal([config["network"]["out"]])))
  return biases
```

Listing 4.1: Automatic creation of biases

Since the biases are only applied to neurons, there is no need to create one for the input layer. While looping through the *layer* array in the configuration, the function creates a variable with random numbers. The amount of numbers is the amount of neurons in the layer. It returns:

[input variables, neurons in layer 1, n. in layer 2, . . . , n. in layer n, output variables]

Similar to listing 4.1 weight vectors are autonomously initialized. To achieve this, a function creates an array with the following structure: This array is used in the following code as *tmpWeightsmodeler*.

```
def createWeights(config):
  weigths = []
  t = 1
  tmpWeightsmodeler = createTmpWeightmodeler(config)
  for x in range(1, (len(config["network"]["layer"])+1)):
      weigths.append(tf.Variable(tf.random_normal([tmpWeightsmodeler[x-1],
          tmpWeightsmodeler[x]])))
      t += 1
  weigths.append(tf.Variable(tf.random_normal([tmpWeightsmodeler[t-1],
      tmpWeightsmodeler[t]])))
  return weigths
```

Listing 4.2: Automatic creation of weight vectors

While *createBiases* created just a vector with one row, *createWeights* creates a tensor with the shape of *e.g. [inputs, neurons_layer_1]*. This needs to be done since weights are applied on the connections between layers.

The biases and weights vectors are used in the creation of the layers. Listing 4.3 is a function to build a simple multilayer feedforward perceptron model.

```
def multilayer_perceptron(x, config, weights, biases):
  layers = []
  lenL = len(config["network"]["layer"])
  lay = tf.add(tf.matmul(x,weights[0]), biases[0])
  layers.append(activationFunction(config, lay, 0))
  for v in range(1, (lenL)):
      lay = tf.add(tf.matmul(layers[v-1],weights[v]), biases[v])
      layers.append(activationFunction(config, lay, v))
  out =  tf.matmul(layers[lenL-1],weights[lenL]) + biases[lenL]
  return layers,out
```

Listing 4.3: Automatic creation of a multilayer perceptron model

First, the placeholder x is multiplied with the first weights-vector to simulate the first connection between the input data with the first hidden layer. The biases are just

added to the result. Afterwards, the connection is applied to the chosen activation function by the function *activationFunction*. This is the TensorFlow representation of layer 1. Afterwards, this is repeated for every hidden layer but instead of the input placeholder, the previous layer is used. In the end, the output layer is created by simple multiplications and adding the variables. Since this is just a prototype no transfer function is used.

Before training, testing and using the neural network, the corresponding sets need to be created. To avoid memory leaks the function shown in listing B.5 is applied only for one file at a time. It starts by creating an array of the whole dataset or of a certain timeframe. Afterwards, it iterates through this array to generate the correct prediction. Therefore, it uses the in the configuration saved time unit *forecast_sec* which defines how many seconds should be predicted. Therefore, there are two methods to determine the value:

1. Calculate the timestamp and add up one second until a tick is found.
2. Iterate through the array until a dataset which is at least the defined time away is found.

Both techniques do not ensure that the exact forecast timestamp is found, but the next dataset after the given seconds. Moreover, the second one can only be applied to sorted raw data. In the end, the function creates an input array for each row with a timestamp or as mentioned in section 4.1 with separate inputs for each time unit. Furthermore, the calculated predictions are combined in a second array.

Listing B.5 and all other descibed functions are finally used within the training function shown in listing B.4. After creating all inital values and the network model, a *cost* and *optimizer* function is implemented. While the *multilayer_perceptron* builds the feed-forward part of the neural network, the *GradientDescentOptimizer* builds the backpropagation. A detailed description of the *Gradient descent* can be found in [6, pp.1765f, 48, p.1231]. The *cost* variable is a way to determine the error of the training iteration. The resulting error is also known as *loss*. This prototype is using a quite simple one, which is mathematically expressed as:

$$(prediction_i - price_t)^2 \tag{4.3.1}$$

The *optimizer* uses this to automatically minimize the error and, therefore, training the NN. Both variables are used within the TensorFlow *session*. In this session, the training data is fed into the *optimizer* in batches.

After all epochs are finished, the trained model is tested with the testing data. To classify whether the prediction is correct, the application subtracts the correct value from the predicted value and checks whether the result is within a 5 error radius. If the percentual change is predicted the error radius is 0.0001. Moreover, for this forecasting, the accuracy of predicting the trend (negative or positive) is calculated. In the end, the trained and tested model is saved on the storage using the *tf.train.Saver* function.

4.4. Influencer Gathering

Like already mentioned in the introduction to this chapter, the prototype is only based on historical data within the time series. Another aspect is to use real-time information to influence the result. As an input for the current mood within the society, this thesis is evaluating real-time tweets from twitter. This should assure that it has not only old but current information about the market. The following section describes how influencer information are gathered and analyzed.

4.4.1. Architecture

The influencer is based on a combination of different structures in the cloud. The base is a NodeJS server which combines the logic, database management, front end, and API. The decision for the open-source, cross-platform JavaScript run-time environment is justified by its simplicity and versatility. It is built on an asynchronous event-driven I/O (input/output) system that guarantees an easily scalable web server capable of handling a large number of simultaneous connections [66]. According to the Foundation [31] the "project is jointly governed by a Technical Steering Committee (TSC) which is responsible for the high-level guidance of the project". They are supported by various working groups which assure a continuous development.

The selected database structure is a MySQL. Although NoSQL databases are widely used nowadays, MySQL is built on top of a relational database schema which is on the market since 1994. As the name already suggests it uses the ***S**tructured **Q**uery **L**anguage*. In comparison to other NoSQL solutions, MySQL has a larger community. Additionally, the confidence in the solution is very high since Oracle is the developer [16]. A lot of NoSQL solutions are community-driven and relatively new on the market. Although some products like MongoDB or Cloudant are currently under development of big companies and are rapidly growing, recent years showed that those can be discontinued very fast. The influencer gathering information is depending on its database since it saves every information in it. Therefore, MySQL is used. A more detailed comparison of MySQL and other NoSQL solutions can be found in [86, 35, 11, 23].

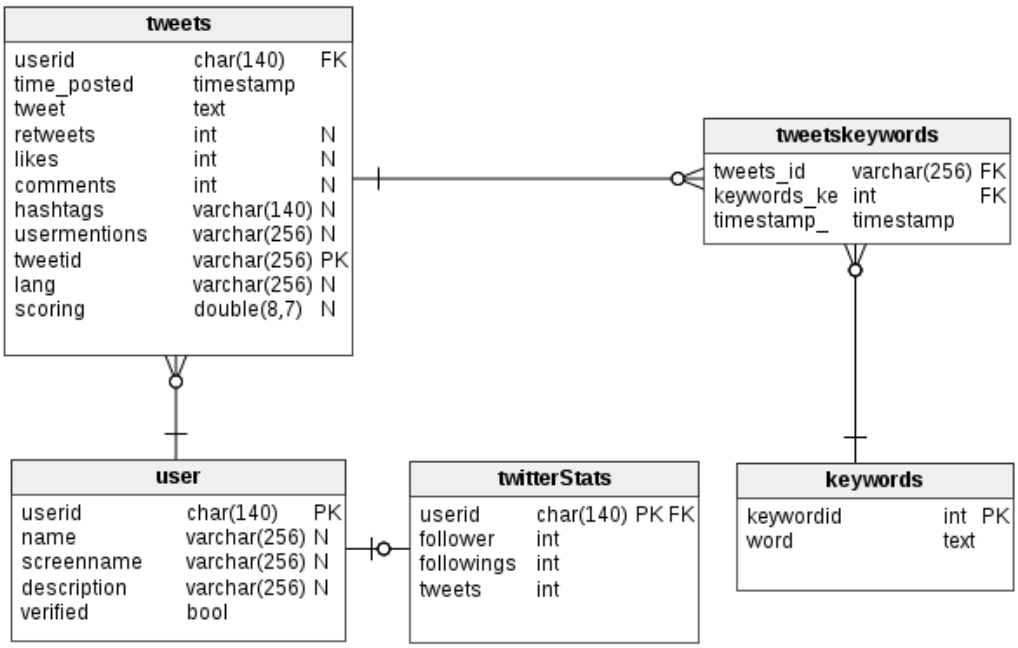


Figure 4.2.: MySQL schema of twitter influencer database

Figure 4.2 shows the used relational database schema. The database is normalized by reference to the *Boyce–Codd normal form* which is further examined in [30]. The only exception is the *twitterStats* table which could be included in the *user* table. It was outsourced to improve the performance of the code. Another used service is the *Watson™ Tone Analyzer* by IBM which is using "linguistic analysis to detect emotional, social, and language tones in written text" [45]. Figure 4.3 shows the basic usage of the service. Within the influencer, tweets are passed and receive a JSON object

with analyzed information. This *tone* document is later used to score the tweet (see section 4.4.2).

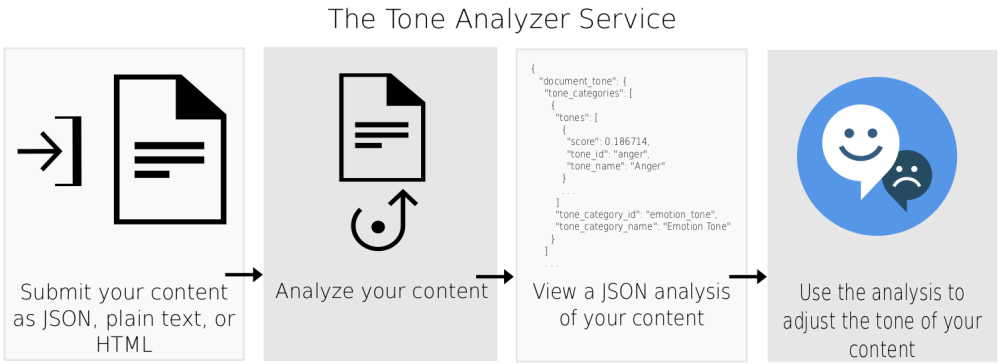


Figure 4.3.: Watson Tone Analyzer flow of calls [45]

The major component of the influencer is the *Twitter streaming API*. The online platform provides different accessing points to their tweets and users. The streaming API suits this thesis the best since it automatically sends new public tweets to the project. The NodeJS package *twit* provides an instance that can be used to make requests to Twitter's APIs (streaming and REST). The detailed implementation can be found in section 4.4.2.

The cloud environment used is IBM *Bluemix*. Although there are different other providers like Amazon or HP, Bluemix is the only platform which provides the Watson services. It supplies scalable instances by "high performance compute and storage infrastructure in secure IBM Cloud Data Centers" [47]. It supports all of the used services (MySQL by Compose, NodeJS, Watson Tone Analyzer) and enables a stable runtime.

4.4.2. Features

The influencer has various features which enable gathering specific tweets and filter, analyze and score those. Firstly, each instance of the program is configurable with various config files:

DB.json specifies external Mysql database credentials. If the Bluemix instance has a MySQL service connected, this file is not used.

CreateDB.sql saves the database structure shown in fig. 4.2. If the database is empty the influencer code automatically creates the model and relationships.

config.json specifies global variables like the static base path for the NodeJS application.

twitter.json contains the credential information for the Twitter API.

whitelist.json is an array of keywords which specifies the tweets which will be gathered. Every time a whitelist is initialized new keywords are added to the database.

blacklist.json specifies certain keywords which filter out tweets.

scoring.json contains information on how to score the tweets.

One major feature is the database implementation which consists of several functions to save, update or select information. Section 4.4.1 already mentioned the general structure. In more detail the two main tables are *user* and *tweets* which are saving every filtered user and tweet with an $1 - n$ relationship. Each tweet is analyzed for certain keywords (which are defined in the whitelist) when it is received. Since each tweet can contain more than one keyword and each keyword can be used in different texts, the relationship $n - m$ is resolved to an intersect table *tweetskeywords*. This table contains an extra timestamp which can be used to identify when the tweet has been analyzed.

The following features will be explained with reference to fig. 4.4 which shows the flow of a tweet through the different functions.

The twitter stream API is capable of automatically filter tweets which are useful for the API user. Therefore, each instance can track up to 400 keywords [18]. In addition, the API is capable of locating tweets based on users or locations but this is not relevant for this application. After initiating the *twit* package with the given credentials in the *twitter.json* config file, the influencer uses the keyword array in the *whitelist.json* to search for certain tweets.

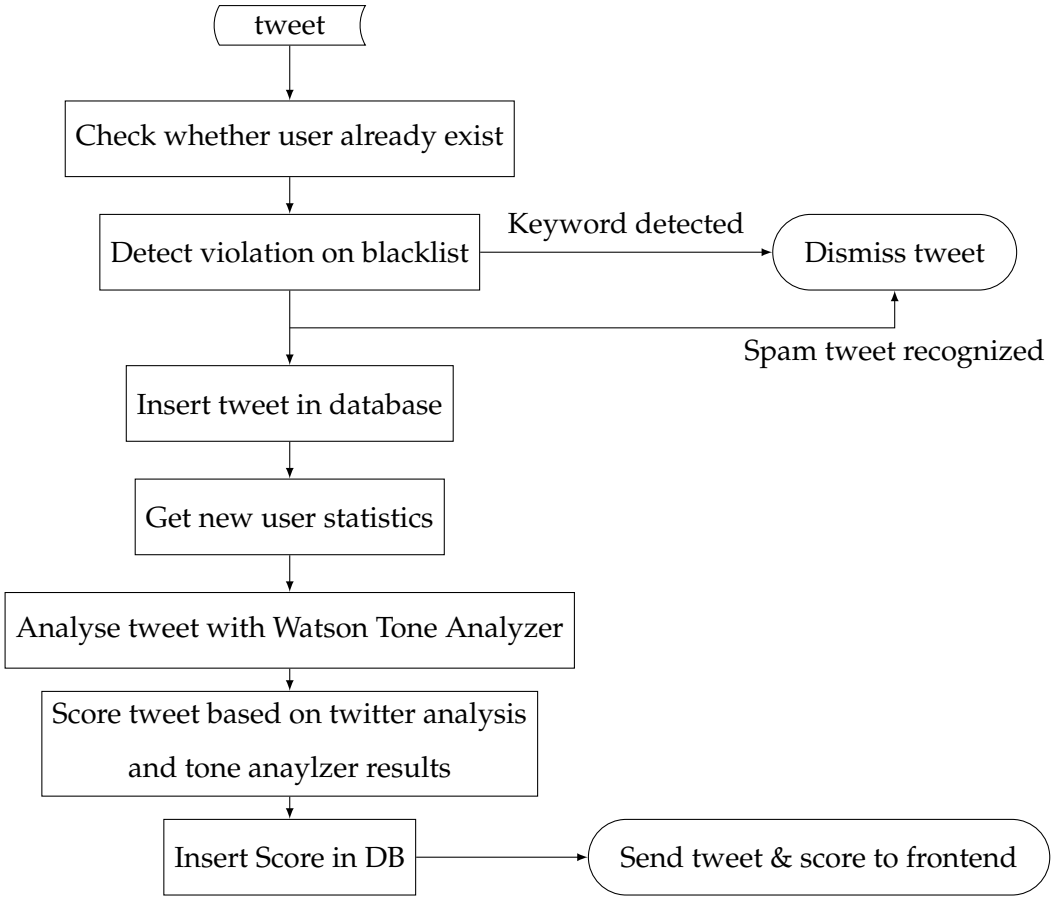


Figure 4.4.: Flow of a tweet in the influencer code

If one of the keywords is found in a tweet, the API sends it automatically to the influencer instance. This starts a sequence of different functions. Such a tweet is represented as a JSON document containing several information shown condensed in listing 4.4.

```
{
  "created_at": "Tue Aug 15 22:47:38 +0000 2017",
  "id": 897590634432929800,
  "id_str": "897590634432929793",
  "text": "Test #IBM #twitterInfluencer #Bachelorarbeit",
  "user": {...},
  "entities": {
    "hashtags": [{
      "text": "Bachelorarbeit", ...
    }, ...],
    "user_mentions": [],
    "media": [...],
    "favorited": false,
    "retweeted": false,
```

```
    "possibly_sensitive": false,
    "filter_level": "low",
    "lang": "eng",
    "timestamp_ms": "1502837258333", ...
}
```

Listing 4.4: Condensed Twitter example tweet document

Based on this information the influencer checks whether the user already exists. If he does, a function uses the information in the *user* object of the tweet to update him. If he does not exists, he is inserted in the database. The implementation updates the user and his statistics, therefore, every time it receives new data from twitter.

Afterwards, the username and text of the tweet is analyzed whether it contains any keywords which are defined in the *blacklist.json* config file. In addition to the tweet's text, especially the hashtags are examined. The config file differs between the user's *name/screen_name* and the tweet itself. Therefore, it is easily possible to ban certain accounts. If a keyword is detected, the tweet is rejected and the procedure exits.

Another filtering technique is used before inserting the tweet in the database. If the user posted any tweets in the last hour, those are compared to the current tweet using NodeJS's *string-similarity* package. If there is a similarity of over 80% the tweet is rejected. If the user posted a tweet with a correlation of over 95% the current tweet is rejected and the old tweet gets reset to a score of 0. This method is preventing spam.

While the tweet is added to the database, another sequence is started to collect the current twitter statistics of the user. This will be used later in the scoring process. Moreover, the keywords in the tweet's text are identified and inserted into the database.

The major feature of the influencer is to evaluate each tweet on its view of the market. This scoring is based on a simple value which represents a positive or bad negative on the market. A neutral mood is represented by a 0 and should not have any impact on the prediction. The maximum positive sentiment is a 1, the worst -1.

To evaluate a tweet, firstly, the text is passed to the Watson Tone Analyzer to receive the linguistic tones in the following categories: (i) *emotion*, (ii) *social*, and (iii) *language*. The detailed explanation of each category and each tone is available at [46]. To understand

the scoring of this analysis results it is imported to understand the *scoring.json* configuration file represented in listing 4.5. A complete example can be found in listing B.7.

```
{
  "scoringweights": {
    "watson": {
      "emotion": {
        "anger": [[0.2, -10]], ...
      }, "social": {
        "agreeableness_big5": [
          [0.5, -0.1],
          [0.75, 0.2]
        ], ...
      }, "language": {
        "analytical": [[0.5, 30]],  ...
      }
    }
  }, "catweights": {
    "watson": 1
  }
}
```

Listing 4.5: Shortened *scoring.json* example

The object *watson* in *scoringweights* contains the information on how each tone of the Watson service is weighted and used. It is always written in the following pattern:

```
  "tone": [
    [threshhold_upper, effect]
  ],
  "tone": [
    [threshhold_lower, effect],
    [threshhold_upper, effect]
  ],
```

If a tone has one array element, on the one hand, it is only an upper threshold. On the other hand, if it has two, the first one is used if a value is below this threshold, the second if a tone value is over the threshold. In general, this means a value has to pass or

stay below a certain boundary to have an influence on the score. The second argument represents the influence of the tone. This is classified in two categories:

> 1 The tone directly affects the scoring.
< 1 The tone effects the complete direct score by a certain percentage.

While keeping this in mind, the Watson score results in a temporary score of -100 to 100 which is later normalized to -1 to 1. Tones, which are greater than 1, multiplicate their score with their own weight. Afterwards, those are summed up. This can be mathematically expressed as:

$$S_{temp}(tweet) = \sum Tone_{>1} * Effect \tag{4.4.1}$$

Otherwise, tones whose effect's impact is below 1 do no directly affect the score. They are added up to a percentage impact, which, in the end, is reinforcing or weakening the temporary score. Therefore, eq. (4.4.1) needs to be supplemented:

$$S_{temp}(tweet) = (\sum Tone_{>1} * Effect) * (1 + \sum Tone_{<1} * Effect)) \tag{4.4.2}$$

In both cases, a positive *Effect* value is boosting the positive part of the sentiment, a negative one boosts the negative mood of the tweet. This quite complex scoring evaluation of the Watson tone is used as some tones directly represent a concrete mood, other only represent *e.g.* a confident level.

Listing 4.5 shows as well the object *catweights*. This represents the weight of each category for the final score of the tweet. Since currently only the Tone Analyzer is implemented, the whole score is based on its results. Although a concrete twitter statistics analysis is not yet added, the influencer program uses this statistics to filter outs extraneous tweets. This is implemented as a relatively easy function, which checks whether the user fulfills all of the following two requirements:

1. The user should at least have ten followers.
2. The user should not have a big discrepancy between followers and people he is following. Currently, this threshold is set to 100. This rule is just used when the user has fewer followers than accounts he is following.

If one of those is not satisfied, the user is at the time of the tweet not relevant for the market, and the tweets score is automatically set to 0.

In the end, the final result is added to the tweet's database entry and is emitted to the front end.

The front end is a simple HTML page which shows some statistics of the database and every tweet which is not filtered out. To achieve a fast efficient page, the NodeJS package *socket.io* is implemented to create a real-time event-based communication.

Each instance is providing a (currently public) API via a REST architecture. The following interfaces are implemented:

[GET] /ping Answers with status 200 and the current timestamp.

[GET] /stats Response is a JSON document with current number users/tweets in the database and the amount of tweets in the last 24 hours.

[GET] /TweetsLastDay Response is a JSON document with the amount of tweets in the last 24 hours and the current timestamp.

[GET] /lastMinute Response is a JSON document with the average score of all tweets (which do not have a neutral score) in the last 60 seconds and the current timestamp.

This application is just a prototype which analyses roughly the influence from the public. There are still some problems with it. For example, while searching for *"stock"*, there are a lot of spam tweets from hijacked accounts posting fake advertisements for goods. Often those tweets contains *"... (article) is back in stock ..."*. One solution for this problem could be analysing every tweet with the *IBM Watson Natural Language Understanding* which is able to categorize texts into different topics like *news*, *international news* or *finance* [44].

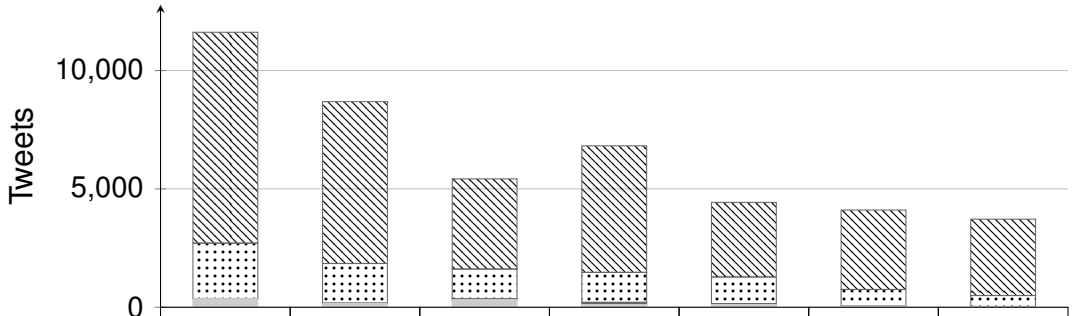
4.4.3. Empirical usage

For a brief overview of the effectivness of the influencer gathering service, it was run one week (*21.08.2017 00:00* to *27.08.2017 23:59:59*). The used whitelist was a mix of different terms within the financial sector to receive a wide varity of results:

["dax", "börse", "DAX", "S&P", "ecb", "finance", "stock"]

The used blacklist is a small example of how certain keywords can prevent spam tweets of entering the database:

{ "username" : ["bot", "freedesignfile"],
"tweet" : ["hiring", "league", "INNINGS", "wicket", "cricket", "fashion"]}



	2017/8/21	2017/8/22	2017/8/23	2017/8/24	2017/8/25	2017/8/26	2017/8/27
Dax	335	142	298	93	92	57	28
Börse	18	11	10	2	10	0	2
S&P	0	0	0	0	0	0	0
ECB	11	21	48	111	45	11	3
Finance	2345	1674	1260	1268	1135	684	462
Stock	8909	6839	3808	5345	3148	3354	3227
Total	32808	28327	16968	23019	12139	19681	8920

Dates

Figure 4.5.: Tweets posted per day

The results shown in fig. 4.5 point out the current issues of the application. The keyword *S&P* is detected zero times through the whole week although a test account posted every keyword on two days within the week. This leads to the issue that the keyword detection is not working properly. Another observation which is assuring this problem is the total amount of tweets in the database in comparison to the total amount of tweets per keyword. On the first day of the test week, there is a total of 32.808 tweets in the database. The total sum of all keyword related tweets is 11.618. This is further encouraged by the fact that one tweet can be related to multiple keywords.

In general, this can have different causes of failure. Some tweets are not written in German or even in the Latin alphabet. Therefore, the simple detection and whitelist are not enough to classify tweets. Twitter is not limited to this subject and therefore sends tweets of all languages which are connected to the whitelist. Especially the not detected keyword *S&P* shows one more point of failure. Special characters seem to be filtered out or not recognized by the influencer implementation.

In the end, the influencer gathering service has some problems so far. Although the first results seem promising, this cannot be used to predict future time series, since there are too many points of failures and not covered special areas of tweets.

5. Testing, performing and Results

The last chapter describes the implementation of the prototype. This chapter is using this implementation for evaluating an optimized configuration for an example time series. Thereby, it describes, performs and interprets a series of experiments.

While using the neural network, the generation of training and testing sets appeared to be a bottleneck. Therefore, the preparation of the correct values for the predicting is outsourced to create a local folder called *data-(Name)/set-(seconds)*, whereby *name* is the name of the raw data set and *seconds* is the time which should be predicted. Similar to the creation of the daily based raw data, every day is stored in a separate *.h5* file. Therefore after an initial creation of these files, the training of a neural network improved from ≈ 110 minutes to $\approx 50 - 70$ minutes. The whole code can be viewed in listing B.6. The processing of 100 files takes for the *YM* example about 280 to 390 seconds when using the second calculation method.

Another optimization is the implementation of multitasking which enables the application to train multiple models separately at the same time.

Experiments

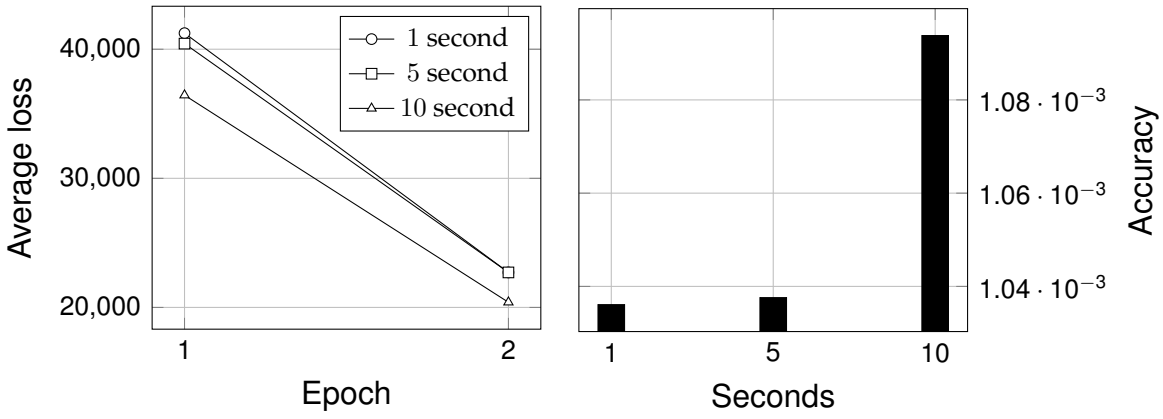
As already described in chapter 4 the neural network prototype is fully configurable. Combining just the most important parameters (*forecast_sec, training_epochs, amount of hidden layers and amount of neurons in each layer*) result in a huge number of possible solutions. Since a training epoch is relatively cost-intensive, this thesis will only evaluate three versions of each parameter one after another. For each step, the following factors are observed and compared:

- Average loss per epoch (see eq. (4.3.1))
- Accuracy of the test set

The average loss gives a good overview on how the model's training precision evolved over the different epochs. Since it is a squared error, the square root of each value shows the real derivation. The accuracy represents the precision of the final modeled. Since this thesis is using a relatively small error, the accuracy reveals whether the NN is usable in a real-life scenario.

The complete *YM* data set is split into 70% training and 30% testing set all the time. The *learning_rate* is set to 0.01 to avoid an overfitting and the *batch_size* is 10.000. Empirical non-recorded usage of the application showed that it is more precise and has less loss if the timestamps of each data set are parsed into their six basic parts and given as separate inputs (*parseDate* is, therefore, set to true).

The first parameter is *forecast_sec*. Since there are no other optimal parameter so far, the other parameter are chosen randomly: (i) *training_epochs* as 2, (ii) *amount of hidden layers* as 3, and (iii) *neurons in each layer* as $112, 50,$ and10. The prototype should be apble to predict short-term trends. Therefore, it does not need to predict a large time gap. The chosen values are 1, 5, and 10.



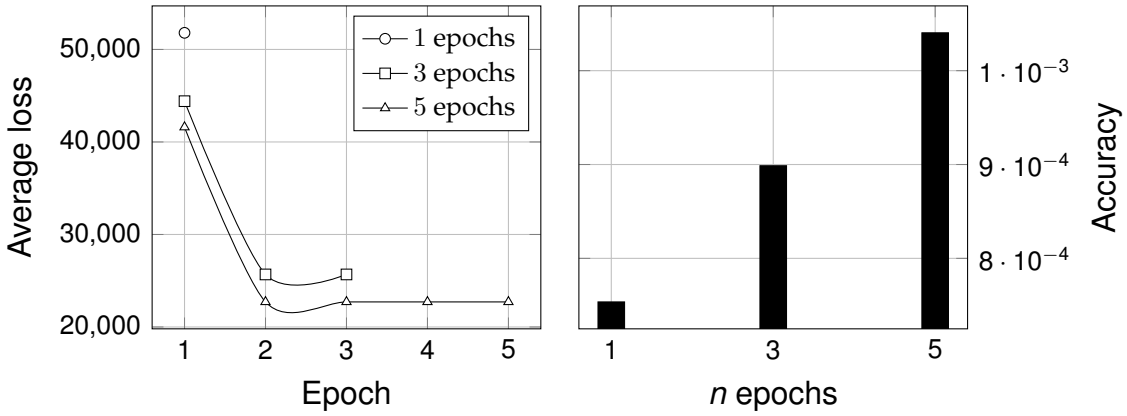
(A) Average loss (B) Accuracy of testing set

Figure 5.1.: Results of different *forecast seconds*

The results shown in fig. 5.1 are quite bad. The best results are made by a 10 seconds forecasting with an accuracy of 0.109%. The other two options are approximated the

same with around 0.104%. There is a good improvement of the loss from epoch 1 to the second one in all cases.

As already a big improvement of the lost between the epochs during the last experiment has been observed, it is the logical step to analyze the influence of this parameter. Since 10 seconds forecasting gives the best result, it is the value for *forecast_sec* in this experiment. The other random configurations stay the same. The test variables for *training_epochs* are 1, 3 and 5.

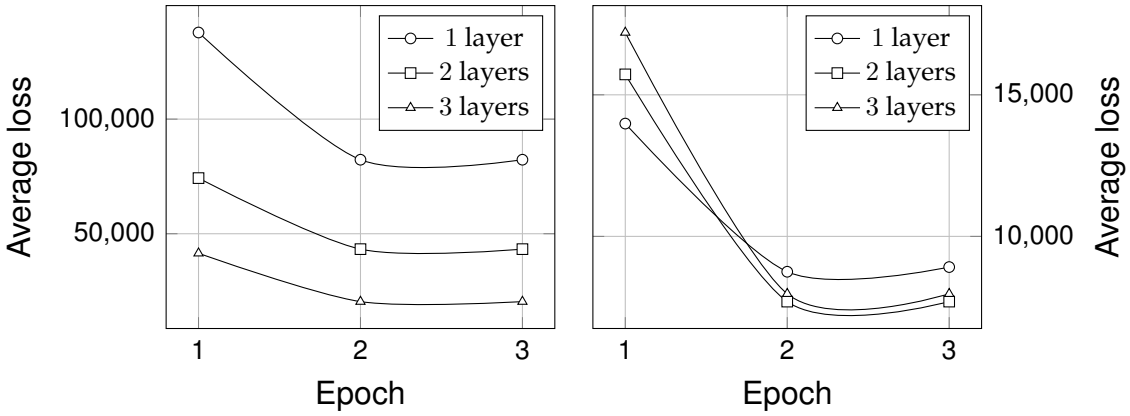


(A) Average loss (B) Accuracy of testing set

Figure 5.2.: Results of different amounts of *epochs*

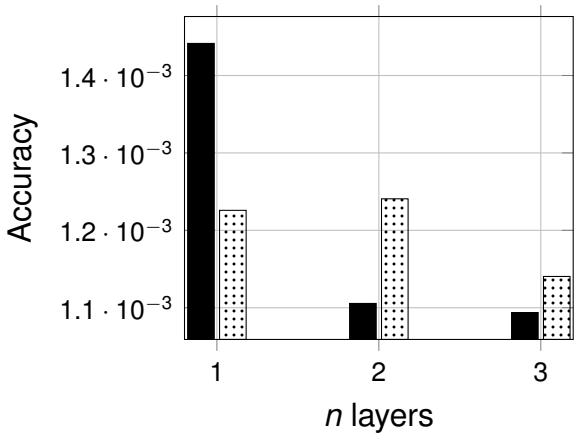
Figure 5.2 shows the result for the second experiment. Although the overall accuracy or average loss has not improved, the loss does not change significantly after the third epoch. Therefore, the optimal training epoch number for this data set is 3.

The third experiment is the iteration of the amount of layers. In the same iteration, it tests the amount of neurons per layer. Therefore, there are two times three tested parameters:(i) 1, 2 and 3 layers with each 10 neurons, and (ii) 1, 2 and 3 layers with each 50 neurons. The *training_epochs* are set to 3 and *forecast_sec* to 10.



(A) Average loss (10 neurons)

(B) Average loss (50 neurons)



(C) Accuracy of testing set (dotted: 50 neurons)

Figure 5.3.: Results of different amounts of *hidden layers* with different number of neurons per layer

As can be seen in fig. 5.3 the variation with more neurons performs better than with only 10. A three hidden layer model with ten units per layer has its best average loss in the third epoch with 20399.1741. In comparison, the 50 model per layer model has its best point at around 7980 with a two and three layer neural network. This is an improvement of over 60%.

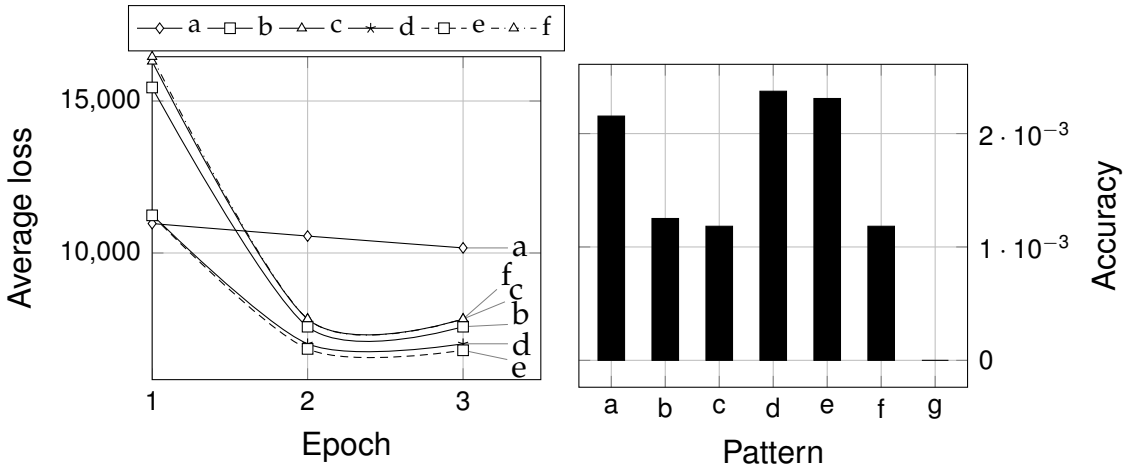
In contrast the accuracy is best within a one layer model with 10 neurons (see fig. 5.3C). Since the accuracy is very low and the difference not that big, this can be seen as a margin of error within the experiment.

The fourth experiment dives deeper into the examination of the optimized amounts of neuron per layer. As already observed in the last experiment, more neurons perform

better. Therefore, this test is using at least 50 neurons. It will explore seven different combinations to review different pattern shown in table 5.1. As can be viewed in the table, this experiment uses a three layer approach. The remaining parameters are the same as in the last attempt.

	Layer 1	Layer 2	Layer 3
a	50	100	150
b	150	100	50
c	50	100	50
d	100	50	100
e	100	100	100
f	500	100	50
g	50	100	500

Table 5.1.: Hidden layer pattern experiments



(A) Average loss (B) Accuracy of testing set

Figure 5.4.: Results of different hidden layern patterns

Pattern **g** was not able to produce any results. Figure 5.4 shows the results of the remaining patterns. The accuracy of the testing set is divided into two groups around

0.001 and 0.002, whereby, the better group doubles the accuracy in comparison to the previous experiments. Nevertheless, even those represent just a correctly predicted 0.2% of the dataset. Figure 5.4A shows that after the second epoch five of the six patterns are in the same range of an average loss. Only **a** is a discordant value. Despite the worst loss, the pattern has the third best accuracy. The two best accuracies have also the smallest average loss.

Derived from all experiments, this is optimized configuration:

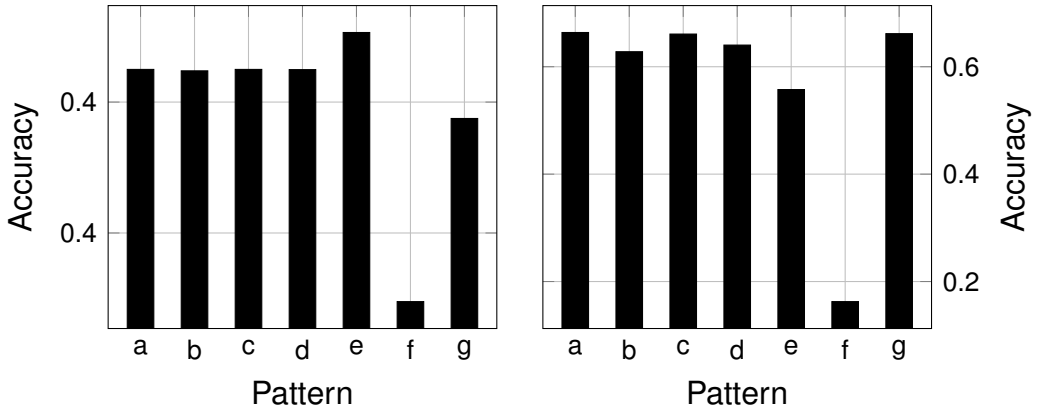
- **forecast_sec**: 10
- **training_epochs**: 3
- **amount of hidden layers**: 3
- **first hidden layer neurons**: 100 or 50
- **second hidden layer neurons**: 50 or 100
- **third hidden layer neurons**: 100 or 150

Even this configuration has a unacceptable prediction accuracy and is not able to be used with the *YM* data in a real-life scenario.

One training epoch took around one hour on the test machine described in section 4.2. Due to the tight time schedule, every experiment was just run once. Therefore, the optimal configuration described in this section is just an evaluation whether a prediction is within the realms of possibility.

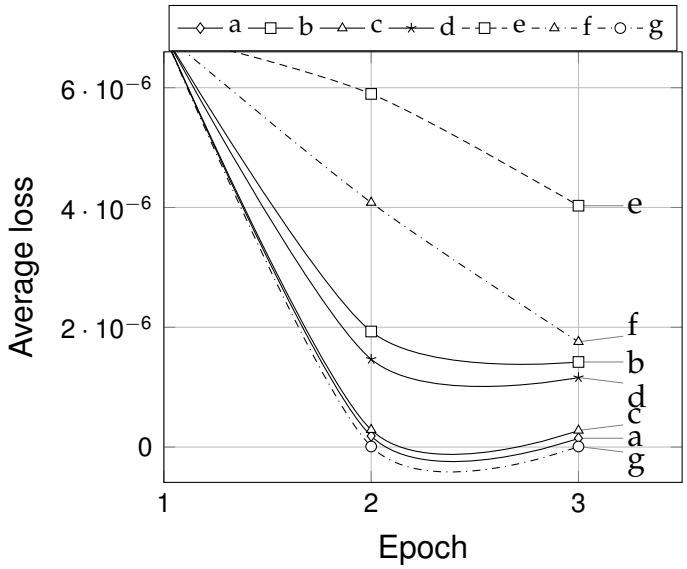
The four experiments result in an optimized configuration but also in a not usable neural network model. Nevertheless, this configuration can be used in a second approach for predicting the time series. By not forecasting an accurate price value, the network can also be used to predict the percental change. This value can give information about whether the TS is rising or falling within the predicted time lag.

To test whether this approach performs better than the first one, the last experiment is repeated with the *predictPercChange* parameter set to true. The patterns are the same as shown in table 5.1.



(A) Accuracy of testing set (percental change)

(B) Accuracy of testing set (trend)



(C) Average loss

Figure 5.5.: Results of predicting the percental change

Figure 5.5 visualizes the accuracy of the trend, as well as from the percentual change, and the average loss for all seven patterns. Since the forecasting is mostly taking place within a range of 0 to 1, the loss is significantly smaller than in the other experiments. Furthermore, the cost calculation is not optimized for this kind of values.

In general, all patterns except **f** are performing similar. Nevertheless, **a** (1.4693×10^{-7}), **c** (2.7746×10^{-7}) and **g** (9.3159×10^{-9}) have the smallest loss. This suggests that the optimized input pattern for this specific problem is an increasing of the amount of neurons for each layer. Since **g** performed better than the other two, it can be reasonably assumed, that actually an exponential increment is the optimal solution.

Hence, the most interesting behavior has pattern **g**. The first epoch has an average loss of around 310.6288. The second training round improves this by multiple times to $\approx 9.3160 \times 10^{-9}$. The last epoch does not change this improvement significantly.

With an accuracy of around 65% for predicting whether the time series will increase or decrease, the NN model is able to be better than throwing a coin each time ($50/50$ chance).

6. Summary

The last chapters intensively described the theoretical background and a basic implementation for a prediction prototype. After accomplishing several experiments, this chapter discusses the results and gives a brief overview of future directions. Furthermore, the complete thesis is summarized and a conclusion of the project is drawn.

6.1. Discussion

This thesis tried to theoretically and practically develop a neural network which is able to predict upcoming short-term trends within a financial time series. The approach was to implement a configurable, easy to modify prototype which is able to automatically train and test different neural networks. In an example scenario based on the Mini Dow Jones, we evaluated an optimized configuration for predictions which can be found on chapter 5.

Although the first experiments in chapter 5 do not seem very promising, the last one results in an accuracy above 50%. This can possibly be explained by the different prediction goals of the two approaches. A financial time series, especially the example one used in this thesis, has different seasonal trends, an inflation rate and an overall growth of the market segment. Hence, the first price can be different than the latest one, although the real value of the asset never changed. Therefore, a precise price prediction is far more complex than a relative (percentual) change rate.

Although the best accuracy is 66%, the prototype is very unlikely to be able to be competitive in a real-life scenario. The precision is based on seven years of data. Hence, the prediction can perform a long time worse than the market and lose all capital within

a short time frame. Even if the predictor is performing fine, additional expenses like transaction costs and the inflation are not embedded in the calculation. Section 2.1 show that those factors combined with the overall market growth can significantly shrink the final profit. Moreover, such an autonomous system has, even more, expenses than a human trader: (i) acquisition costs of the hardware, (ii) electricity expenses since the system needs to run all the time, and (iii) administration costs to keep the system running or to implement enhancements. Therefore, an accuracy of around 60% is just too small to justify the involved risk.

Nevertheless, the concept of neural networks is very powerful. The variety of possible configurations combined with the data independent structure has a lot of potential fields of application. Despite this capability, it needs to be carefully evaluated if a NN solution makes sense. The cost-intensive training is a huge overhead if traditional analysis methods also yield to an acceptable solution.

6.2. Future Directions

Section 4.3 describes only a basic prototype of a neural network with mixed results. As described in the theoretical part of this thesis, there are various options which can be implemented to improve the prediction accuracy.

First of all, one problem is, that the implemented neural network never use predicted data to improve its model. This leads to the problem, that if it trains *e.g.* a data set from 2009 to 2013 and is testing the whole year 2014, the last prediction in the testing set is based on one-year-old data. This leads to a no acceptable time gap for the most predictions.

For example, one flaw of the prototype is that it uses raw input data instead of using aggregated candlestick sets. This could reduce the noise and prevent an overfitting of the network model. Furthermore, time series analyses methods like the moving average could improve the performance of the application. Some other enhancements could be:

Markov chains Automate the application to generate multiple NNs with different window sizes. These can now be used to compare different methods. In the end, those could automatically evaluate against each other based on how many predictions were correct or which predicted the best result. A resulting score could be used to switch between models automatically or to return a weighted average result.

Varity A more widely experiment could compare even more network configurations than already done on chapter 5. While analysis those outcomes, it is possible to achieve better results. This *try-and-error* approach could reveal hidden examples to achieve a better accuracy.

Input The prototype only uses one specific time series. When evaluating more and different input data, the results could vary since each TS has different information.

Real-time data Currently the prototype is using only historical data. The logical next step is to connect it to a real-time API to predict future values.

Influencer data In section 4.4 a rough implementation of an influencer gathering service is described. After solving its bugs and refining its configuration, it could be used to connect more real-time data to the neural network.

Performance Since the presented neural network prototype is only a proof of concept, it is not very optimised. When running the application in a real-life scenario, new knowledge is gained which could improve its runtime performance.

The prototype presented in this thesis has a very simple and basic structure. Nonetheless, it had to be tested on a powerful machine. Since this high-performance costs are of greatest disadvantages from neural networks, Intel and Movidius (Intel company) have recently done a completely different step to improve the use of NNs. In late July 2017 they launched the *Neural Compute Stick* with a *Myriad 2 VPU*. Accordingly, to Remi El-Ouazzane, vice president and general manager of Movidius, the USB 3.0 stick is capable of calculating "more than 100 gigaflops of performance within a 1W power envelope" [63]. The unit is based on an embedded neural network which can process an automatically converted *Caffe*-based convolutional neural network (CNN).

The features of the device are: **compiling** the network, **tune** the model to optimal real-world performance and **accelerate** by "adding dedicated deep learning inference

capabilities to existing computing platforms" [63]. The stick is addressed to developers and scientists to expand their local workstation without an external network connection. In combination with a small compute unit, like a Raspberry Pi, it can also be used to create a local NN worker. In the future, the compute unit should also upgrade IoT devices or machines to be more "intelligent". [63] Moreover, it solves many flaws of the usage of neural networks for predicting financial time series. Acquisition and electricity expenses shrink to an absolute minimum.

6.3. Conclusion

All in all, this thesis was mostly a success. The extensive theoretical part examines the most important techniques to analyze and forecast a time series. Furthermore, it gives a brief insight to the financial world, while describing in detail the concepts and methods of neural networks. The practical segments evaluate and develop the best implementation for a prototype. Therefore, a detailed series of experiments was accomplished to find the optimal configuration for a market segment.

As section 6.1 already pointed out, the prototype of this thesis is able to predict short-term upcoming trends (rising or falling in the near future). Although this proof of concept is usable, it was not possible to build an autonomic training algorithm in the given time. The complexity of neural networks and data preparation methods is massive.

In general, neural networks have a bright future. With upcoming new technologies like the Movidius Neural Compute Stick, even small devices are able to be more intelligent. Newer, simpler frameworks like TensorFlow or Caffee2 will accelerate this process. Even though a lot of use cases are based on real-time usages like live speech or handwriting understanding, NN models are also able to predict the future. Whether this will bring us closer to traveling back in time, is a question for another day.

Bibliography

[1] Ratnadip Adhikari and R. K. Agrawal. "A combination of artificial neural network and random walk models for financial time series forecasting". In: *Neural Computing and Applications* 24.6 (2014), pp. 1441–1449. ISSN: 1433-3058. DOI: 10.1007/s00521-013-1386-y. URL: https://doi.org/10.1007/s00521-013-1386-y.

[2] I. Aleksander and H. Morton. *An introduction to neural computing*. Chapman and Hall, 1990. ISBN: 9780412377808. URL: https://books.google.co.uk/books?id=b4ZQAAAAMAAJ.

[3] Hossein Arsham. *Time Series Analysis for Business Forecasting*. 2015. URL: http://home.ubalt.edu/ntsbarsh/Business-stat/stat-data/Forecast.htm#rhowtrend (visited on 08/20/2017).

[4] Bruno Baruque. *Fusion Methods for Unsupervised Learning Ensembles (Studies in Computational Intelligence)*. Springer, 2010. ISBN: 3642162045. URL: https://www.amazon.com/Unsupervised-Learning-Ensembles-Computational-Intelligence/dp/3642162045?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=3642162045.

[5] Andrew Beattie. *The History of Stock Exchanges*. 2017. URL: http://www.investopedia.com/articles/07/stock-exchange-history.asp (visited on 08/21/2017).

[6] "Gradient Descent". In: *Encyclopedia of Neuroscience*. Ed. by Marc D. Binder, Nobutaka Hirokawa, and Uwe Windhorst. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1765–1766. ISBN: 978-3-540-29678-2. DOI: 10.1007/978-3-540-29678-2_2075. URL: https://doi.org/10.1007/978-3-540-29678-2_2075.

[7] George E. P. Box, Gwilym M. Jenkins, and Gregory C. Reinsel. *Time Series Analysis: Forecasting and Control*. Wiley, 2008. ISBN: 0470272848. URL: https://www.amazon.com/Time-Analysis-Forecasting-George-Box/dp/0470272848?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0470272848.

[8] George Bragues. "The Stock Market". In: *Money, Markets, and Democracy: Politically Skewed Financial Markets and How to Fix Them*. New York: Palgrave Macmillan US, 2017, pp. 119–176. ISBN: 978-1-137-56940-0. DOI: 10.1057/978-1-137-56940-0_5. URL: https://doi.org/10.1057/978-1-137-56940-0_5.

[9] Pierre Bremaud. *Markov Chains - Gibbs Fields, Monte Carlo Simulation, and Queues*. Berlin Heidelberg: Springer Science & Business Media, 2013. ISBN: 978-1-475-73124-8.

[10] J. Burton Browning and Marty Alchin. "Principles and Philosophy". In: *Pro Python: Second Edition*. Berkeley, CA: Apress, 2014, pp. 1–18. ISBN: 978-1-4842-0334-7. DOI: 10.1007/978-1-4842-0334-7_1. URL: https://doi.org/10.1007/978-1-4842-0334-7_1.

[11] Craig Buckler. *SQL vs NoSQL: The Differences*. 2015. URL: https://www.sitepoint.com/sql-vs-nosql-differences/ (visited on 08/21/2017).

[12] Deutsche Bundesbank. *Entwicklung des DAX in den Jahren von 1987 bis 2016*. 2017. URL: https://de.statista.com/statistik/daten/studie/199158/umfrage/jaehrliche-entwicklung-des-dax-seit-1987/ (visited on 07/17/2017).

[13] CESifo-Gruppe. *Gemeinschaftsdiagnose: Inflationsrate in Deutschland von 2008 bis 2016 sowie Prognose bis 2018 (Veränderung gegenüber Vorjahr)*. 2017. URL: https://de.statista.com/statistik/daten/studie/5851/umfrage/prognose-zur-entwicklung-der-inflationsrate-in-deutschland (visited on 07/17/2017).

[14] François Chollet et al. *Keras*. https://github.com/fchollet/keras. 2015.

[15] Cook & Bynum. *Determining Intrinsic Value*. 2013. URL: http://www.cookandbynum.com/our-investment-approach/how-we-think/determining-intrinsic-value/ (visited on 07/17/2017).

[16] Oracle Corporation. *MySQL*. 2017. URL: https://www.mysql.com/de/ (visited on 08/21/2017).

[17] scikit-learn developers. *Frequently Asked Questions — scikit-learn 0.18.2 documentation*. 2017. URL: http://scikit-learn.org/stable/faq.html#will-you-add-gpu-support (visited on 08/11/2017).

[18] Twitter Developers. *POST statuses/filter*. 2017. URL: https://dev.twitter.com/streaming/reference/post/statuses/filter (visited on 08/21/2017).

[19] Sander Dieleman et al. *Lasagne: First release.* 2015. DOI: 10.5281/zenodo.27878. URL: http://dx.doi.org/10.5281/zenodo.27878.

[20] Thomas W. Dinsmore. "Machine Learning". In: *Disruptive Analytics: Charting Your Strategy for Next-Generation Business Analytics*. Berkeley, CA: Apress, 2016, pp. 169–198. ISBN: 978-1-4842-1311-7. DOI: 10.1007/978-1-4842-1311-7_8. URL: https://doi.org/10.1007/978-1-4842-1311-7_8.

[21] Christian L Dunis, Jason Laws, and Ulrike Schilling. "Currency trading in volatile markets: Did neural networks outperform for the EUR/USD during the financial crisis 2007–2009?" In: *Journal of Derivatives & Hedge Funds* 18.1 (2012), pp. 2–41. ISSN: 1753-965X. DOI: 10.1057/jdhf.2011.31. URL: https://doi.org/10.1057/jdhf.2011.31.

[22] S.W. Ellacott. "Techniques for the mathematical analysis of neural networks". In: *Journal of Computational and Applied Mathematics* 50.1 (1994), pp. 283 –297. ISSN: 0377-0427. DOI: http://dx.doi.org/10.1016/0377-0427(94)90307-7. URL: http://www.sciencedirect.com/science/article/pii/0377042794903077.

[23] DB Engines. *Cloudant vs. MongoDB vs. MySQL Comparison*. URL: https://db-engines.com/en/system/Cloudant%3BMongoDB%3BMySQL (visited on 08/21/2017).

[24] Bradley J. Erickson et al. “Toolkits and Libraries for Deep Learning”. In: *Journal of Digital Imaging* 30.4 (2017), pp. 400–405. ISSN: 1618-727X. DOI: 10.1007/s10278-017-9965-6. URL: https://doi.org/10.1007/s10278-017-9965-6.

[25] Stefan Etschberger et al. “The Classification of Candlestick Charts: Laying the Foundation for Further Empirical Research”. In: *From Data and Information Analysis to Knowledge Engineering: Proceedings of the 29th Annual Conference of the Gesellschaft für Klassifikation e.V. University of Magdeburg, March 9–11, 2005*. Ed. by Myra Spiliopoulou et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 526–533. ISBN: 978-3-540-31314-4. DOI: 10.1007/3-540-31314-1_64. URL: http://dx.doi.org/10.1007/3-540-31314-1_64.

[26] European Central Bank. *Definition of price stability*. 2017. URL: https://www.ecb.europa.eu/mopo/strategy/pricestab/html/index.en.html (visited on 07/18/2017).

[27] Ludwig Fahrmeir, Thomas Kneib, and Stefan Lang. *Regression: Modelle, Methoden und Anwendungen (Statistik und ihre Anwendungen) (German Edition)*. Springer, 2009. ISBN: 364201836X. URL: https://www.amazon.com/Regression-Modelle-Methoden-Anwendungen-Statistik/dp/364201836X?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=364201836X.

[28] Eugene F. Fama. “Efficient Capital Markets: A Review of Theory and Empirical Work”. In: *The Journal of Finance* 25.2 (1970), pp. 383–417. ISSN: 00221082, 15406261. URL: http://www.jstor.org/stable/2325486.

[29] Eugene F. Fama and G.William Schwert. “Asset returns and inflation”. In: *Journal of Financial Economics* 5.2 (1977), pp. 115 –146. ISSN: 0304-405X. DOI: http://dx.doi.org/10.1016/0304-405X(77)90014-9. URL: http://www.sciencedirect.com/science/article/pii/0304405X77900149.

[30] Flavio Ferrarotti et al. “The Boyce-Codd-Heath Normal Form for SQL”. In: *Logic, Language, Information and Computation: 18th International Workshop, WoLLIC 2011, Philadelphia, PA, USA. Proceedings*. Ed. by Lev D. Beklemishev and Ruy de Queiroz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 110–122. ISBN: 978-3-642-20920-8. DOI: 10.1007/978-3-642-20920-8_14. URL: https://doi.org/10.1007/978-3-642-20920-8_14.

[31] Node.js Foundation. *Project Governance | Node.js*. 2017. URL: https://nodejs.org/en/about/governance/ (visited on 08/21/2017).

[32] Python Software Foundation. *General FAQ — Python 3.6.2 documentation*. 2017. URL: https://docs.python.org/3/faq/general.html#what-is-python (visited on 08/27/2017).

[33] Python Software Foundation. *Python*. 2017. URL: https://www.python.org/ (visited on 08/27/2017).

[34] David Freedman. *Markov Chains* -. Berlin Heidelberg: Springer Science & Business Media, 2012. ISBN: 978-1-461-25500-0.

[35] Ali Gajani. *The key differences between SQL and NoSQL DBs*. 2017. URL: http://www.monitis.com/blog/cc-in-review-the-key-differences-between-sql-and-nosql-dbs/ (visited on 08/21/2017).

[36] Boerse Stuttgart GmbH. *Terms and Conditions Regulated Unofficial Market*. 2017. URL: https://www.boerse-stuttgart.de/en/company/all-about-trading/rules-and-regulations/terms-and-conditions-regulated-unofficial-market/ (visited on 07/21/2017).

[37] GodmodeTrader-Team. *1.7. Candlestickcharts und Candlestick-Muster*. 2016. URL: https://www.godmode-trader.de/know-how/1-7-candlestickcharts-und-candlestick-muster,3733755 (visited on 07/17/2017).

[38] B. Graham. *The Intelligent Investor, Rev. Ed*. HarperCollins, 2009. ISBN: 9780061745171. URL: https://books.google.de/books?id=klYquu4H1wAC.

[39] B. Graham and D.L.F. Dodd. *Security Analysis: The Classic 1934 Edition*. McGraw-Hill Education, 1934. ISBN: 9780070244962. URL: https://books.google.de/books?id=wXlrnZ1uqK0C.

[40] Deutsche Börse Group. *Regulated Market*. 2017. URL: http://deutsche-boerse.com/dbg-en/about-us/services/know-how/glossary/glossary-article/Regulated-Market/2561280 (visited on 07/21/2017).

[41] M.T. Hagan et al. *Neural network design*. Martin Hagan, 2014. ISBN: 9780971732117. URL: https://books.google.de/books?id=4EW9oQEACAAJ.

[42] Jeff Heaton. "Encog: Library of Interchangeable Machine Learning Models for Java and C#". In: *Journal of Machine Learning Research* 16 (2015), pp. 1243–1247. URL: http://jmlr.org/papers/v16/heaton15a.html.

[43] D. O. Hebb. *The Organization of Behavior*. Wiley, New York, 1949.

[44] IBM. *Natural Language Understanding*. 2017. URL: https://www.ibm.com/watson/developercloud/doc/natural-language-understanding/ (visited on 08/24/2017).

[45] IBM. *Watson Tone Analyzer*. 2017. URL: https://console.bluemix.net/docs/services/tone-analyzer/index.html#about (visited on 08/21/2017).

[46] IBM. *Watson Tone Analyzer: General purpose tones*. 2017. URL: https://console.bluemix.net/docs/services/tone-analyzer/using-tone.html#tones (visited on 08/22/2017).

[47] IBM. *What is Bluemix?* 2017. URL: https://console.bluemix.net/docs/overview/whatisbluemix.html#bluemixoverview (visited on 08/21/2017).

[48] Mohamed Ibnkahla. "Nonlinear System Identification Using Neural Networks Trained with Natural Gradient Descent". In: *EURASIP Journal on Advances in Signal Processing* 2003.12 (2003), p. 574805. ISSN: 1687-6180. DOI: 10.1155/S1110865703306079. URL: https://doi.org/10.1155/S1110865703306079.

[49] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

[50] John Maynard Keynes. *The General Theory Of Employment Interest And Money (1936)*. Kessinger Publishing, LLC, 2010. ISBN: 1169831990. URL: https://www.amazon.com/General-Theory-Employment-Interest-Money/dp/1169831990?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1169831990.

[51] Ben Kröse and Patrick van der Smagt. *An introduction to Neural Networks*. 1993.

[52] Jean-Jacques Lambin. "A Stabilized and Regulated Financial Market". In: *Rethinking the Market Economy: New challenges, new ideas, new opportunities*. London: Palgrave Macmillan UK, 2014, pp. 13–28. ISBN: 978-1-137-39291-6. DOI: 10.1057/9781137392916_2. URL: https://doi.org/10.1057/9781137392916_2.

[53] In: *Encyclopedia of Finance*. Ed. by Cheng-Few Lee and Alice C. Lee. Boston, MA: Springer US, 2006. ISBN: 978-0-387-26336-6. DOI: 10.1007/0-387-26336-5_1669. URL: http://dx.doi.org/10.1007/0-387-26336-5_1669.

[54] Svein Linge and Hans Petter Langtangen. *Programming for Computations - Python: A Gentle Introduction to Numerical Simulations with Python (Texts in Computational Science and Engineering)*. Springer, 2016. ISBN: 3319324276. URL: https://www.amazon.com/Programming-Computations-Introduction-Simulations-Computational/dp/3319324276.

[55] P. Malik. *Redesigning the Stock Market: A Fractal Approach*. Response Books. SAGE Publications, 2011. ISBN: 9788132119326. URL: https://books.google.co.uk/books?id=amDlDAAAQBAJ.

[56] B.G. Malkiel. *A Random Walk Down Wall Street: Including a Life-cycle Guide to Personal Investing*. Norton, 1999. ISBN: 9780393047813. URL: https://books.google.de/books?id=v8ENTFP29tkC.

[57] John Mannes. *Facebook open sources Caffe2, its flexible deep learning framework of choice*. 2017. URL: https://techcrunch.com/2017/04/18/facebook-open-sources-caffe2-its-flexible-deep-learning-framework-of-choice/ (visited on 08/07/2017).

[58] Ben R. Marshall, Martin R. Young, and Rochester Cahan. "Are candlestick technical trading strategies profitable in the Japanese equity market?" In: *Review of Quantitative Finance and Accounting* 31.2 (2008), pp. 191–207. ISSN: 1573-7179. DOI: 10.1007/s11156-007-0068-1. URL: http://dx.doi.org/10.1007/s11156-007-0068-1.

[59] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: http://dx.doi.org/10.1007/BF02478259.

[60] Sean P. Meyn and Richard L. Tweedie. *Markov Chains and Stochastic Stability*. Berlin Heidelberg: Springer Science & Business Media, 2012. ISBN: 978-1-447-13267-7.

[61] Adam Milton. *What are Futures? Definition and Examples*. 2016. URL: https://www.thebalance.com/what-are-futures-definition-and-examples-1031172 (visited on 07/25/2017).

[62] M. Minsky and S. Papert. *Perceptrons*. Cambridge, MA: MIT Press, 1969.

[63] Movidius. *Intel Democratizes Deep Learning Application Development with Launch of Movidius Neural Compute Stick*. 2017. URL: https://www.movidius.com/news/intel-democratizes-deep-learning-application-development-with-launch-of-movidius-neural-compute-stick (visited on 08/01/2017).

[64] Ivan Nešić et al. “Modeling Candlestick Patterns with Interpolative Boolean Algebra for Investment Decision Making”. In: *Soft Computing Applications: Proceedings of the 5th International Workshop Soft Computing Applications (SOFA)*. Ed. by Valentina Emilia Balas et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 105–115. ISBN: 978-3-642-33941-7. DOI: 10.1007/978-3-642-33941-7_12. URL: http://dx.doi.org/10.1007/978-3-642-33941-7_12.

[65] J.R. Norris. *Markov Chains*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1998. ISBN: 9780521633963. URL: https://books.google.de/books?id=qM65VRmOJZAC.

[66] Den Odell. “The Node.js Application Platform”. In: *Pro JavaScript Development: Coding, Capabilities, and Tooling*. Berkeley, CA: Apress, 2014, pp. 369–390. ISBN: 978-1-4302-6269-5. DOI: 10.1007/978-1-4302-6269-5_14. URL: https://doi.org/10.1007/978-1-4302-6269-5_14.

[67] Ajoy K. Palit and Dobrivoje Popovic. *Computational Intelligence in Time Series Forecasting: Theory and Engineering Applications (Advances in Industrial Control)*. Springer, 2006. URL: https://www.amazon.com/Computational-Intelligence-Time-Forecasting-Applications-ebook/dp/B000UGR1KG?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B000UGR1KG.

[68] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[69] The pandas project. *Pandas: Python Data Analysis Library*. 2017. URL: http://pandas.pydata.org/about.html (visited on 08/27/2017).

[70] The Money Project. *Größte Börsenbetreiber der Welt nach der Marktkapitalisierung gelisteter Unternehmen im Jahr 2015 (in Milliarden US-Dollar)*. 2017. URL: https://de.statista.com/statistik/daten/studie/166142/umfrage/groesste-boersenbetreiber-nach-marktkapitalisierung-gelisteter-unternehmen/ (visited on 07/05/2017).

[71] “Regression”. In: *Encyclopedia of Genetics, Genomics, Proteomics and Informatics*. Dordrecht: Springer Netherlands, 2008, pp. 1661–1661. ISBN: 978-1-4020-6754-9. DOI: 10.1007/978-1-4020-6754-9_14329. URL: https://doi.org/10.1007/978-1-4020-6754-9_14329.

[72] Geoff Riley. *Index Numbers*. 2016. URL: https://www.tutor2u.net/economics/reference/handling-economic-data-using-index-numbers (visited on 07/25/2017).

[73] Sean Ross. *In economics, what is an index number?* 2015. URL: http://www.investopedia.com/ask/answers/073015/economics-what-index-number.asp?lgl=rira-baseline-vertical (visited on 07/25/2017).

[74] Kristian Rother. *Pro Python Best Practices: Debugging, Testing and Maintenance*. Apress, 2017. URL: https://www.amazon.com/Pro-Python-Best-Practices-Maintenance-ebook/dp/B06XPFQX1X?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B06XPFQX1X.

[75] D. E. Rumelhart and J. L. McClelland, eds. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge, MA: Bradford Books/MIT Press, 1986.

[76] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[77] Ingrid Russell. *The Delta Rule*. URL: http://uhavax.hartford.edu/compsci/neural-networks-delta-rule.html (visited on 08/25/2017).

[78] Renate Sitte and Joaquin Sitte. "Neural Networks Approach to the Random Walk Dilemma of Financial Time Series". In: *Applied Intelligence* 16.3 (2002), pp. 163–171. ISSN: 1573-7497. DOI: 10.1023/A:1014380315182. URL: https://doi.org/10.1023/A:1014380315182.

[79] Facebook Open Source. *What is Caffe2?* 2017. URL: https://caffe2.ai/docs/caffe-migration.html (visited on 08/07/2017).

[80] Investopedia Staff. *Financial Market*. 2017. URL: http://www.investopedia.com/terms/d/derivative.asp (visited on 07/18/2017).

[81] Investopedia Staff. *Futures*. 2017. URL: http://www.investopedia.com/terms/f/futures.asp (visited on 07/25/2017).

[82] Investopedia Staff. *Inflation-Adjusted Return*. 2017. URL: http://www.investopedia.com/terms/i/inflation_adjusted_return.asp (visited on 07/20/2017).

[83] Investopedia Staff. *Option*. 2017. URL: http://www.investopedia.com/terms/o/option.asp (visited on 07/25/2017).

[84] Investopedia Staff. *Security*. 2017. URL: http://www.investopedia.com/terms/s/security.asp (visited on 07/21/2017).

[85] Google Brain Team. *TensorFlow*. 2017. URL: https://www.tensorflow.org/ (visited on 08/09/2017).

[86] Adrian Tear. "SQL or NoSQL? Contrasting Approaches to the Storage, Manipulation and Analysis of Spatio-temporal Online Social Network Data". In: *Computational Science and Its Applications – ICCSA 2014: 14th International Conference, Guimarães, Portugal, June 30 – July 3, 2014, Proceedings, Part I*. Ed. by Beniamino Murgante et al. Cham: Springer International Publishing, 2014, pp. 221–236. ISBN: 978-3-319-09144-0. DOI: 10.1007/978-3-319-09144-0_16. URL: https://doi.org/10.1007/978-3-319-09144-0_16.

[87] TensorFlow. *Installing TensorFlow*. 2017. URL: https://www.tensorflow.org/install/.

[88] TensorFlow. *Python API r.1.3*. 2017. URL: https://www.tensorflow.org/api_docs/python/ (visited on 08/24/2017).

[89] Theano Development Team. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688.

[90] unknown. *Literature review of stock market*. URL: https://nccur.lib.nccu.edu.tw/bitstream/140.119/33947/6/93306406.pdf (visited on 07/21/2017).

[91] R. T. Valeev and A. F. Terpugov. "Calculation of the characteristics of Japanese candlesticks for the samuelson model of price change". In: *Russian Physics Journal* 43.4 (2000), pp. 264–272. ISSN: 1573-9228. DOI: 10.1007/BF02508357. URL: http://dx.doi.org/10.1007/BF02508357.

[92] Paul Whiteley. "Time series analysis". In: *Quality and Quantity* 14.1 (1980), pp. 225–247. ISSN: 1573-7845. DOI: 10.1007/BF00154800. URL: https://doi.org/10.1007/BF00154800.

[93] John B. Williams. *The Theory of Investment Value (Contrary Opinion Library)*. Fraser Publishing Co., 1997. ISBN: 087034126X. URL: https://www.amazon.com/Theory-Investment-Contrary-Opinion-Library/dp/087034126X?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=087034126X.

[94] B. Wu. "An introduction to neural networks and their applications in manufacturing". In: *Journal of Intelligent Manufacturing* 3.6 (1992), pp. 391–403. ISSN: 1572-8145. DOI: 10.1007/BF01473534. URL: http://dx.doi.org/10.1007/BF01473534.

[95] Qingling Wu. *Time Series Analysis*. 2013. URL: http://www2.geog.ucl.ac.uk/~mdisney/teaching/GEOGG121/time_series/GEOGG121_5_TimeSeries_Wu.pdf (visited on 08/20/2017).

[96] Haibin Xie, Kuikui Fan, and Shouyang Wang. "The role of Japanese Candlestick in DVAR model". In: *Journal of Systems Science and Complexity* 28.5 (2015), pp. 1177–1193. ISSN: 1559-7067. DOI: 10.1007/s11424-014-2201-2. URL: http://dx.doi.org/10.1007/s11424-014-2201-2.

[97] Yi C. Zhang and Alexander C. Kagen. "Machine Learning Interface for Medical Image Analysis". In: *Journal of Digital Imaging* (2016). ISSN: 1618-727X. DOI: 10.1007/s10278-016-9910-0. URL: https://doi.org/10.1007/s10278-016-9910-0.

[98] Walter Zucchini, Iain L. MacDonald, and Roland Langrock. *Hidden Markov Models for Time Series - An Introduction Using R, Second Edition*. Boca Raton, Fla: CRC Press, 2016. ISBN: 978-1-482-25384-9.

Appendices

A. Additional Graphs

A.1. Performance indicators by Kröse and Smagt

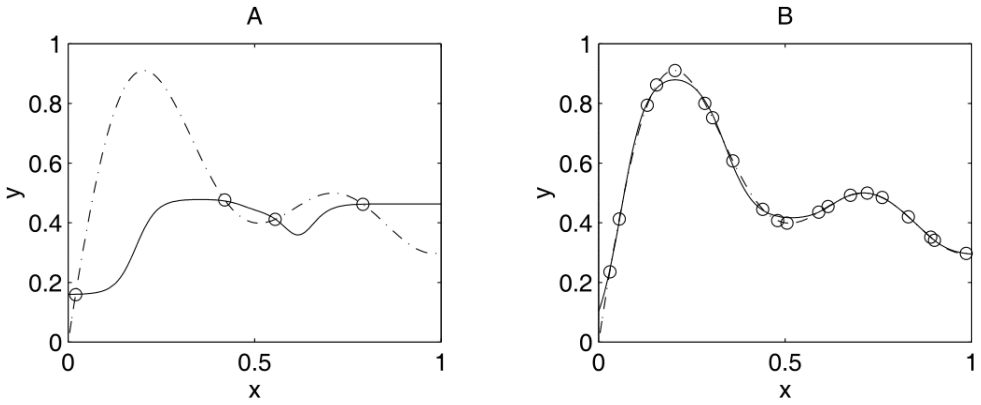


Figure A.1.: "Effect of the learning set size on the generalization. The dashed line gives the desired function, the learning samples are depicted as circles and the approximation by the network is shown by the drawn line. 5 hidden units are used. a) 4 learning samples. b) 20 learning samples." [51, Figure 4.7]

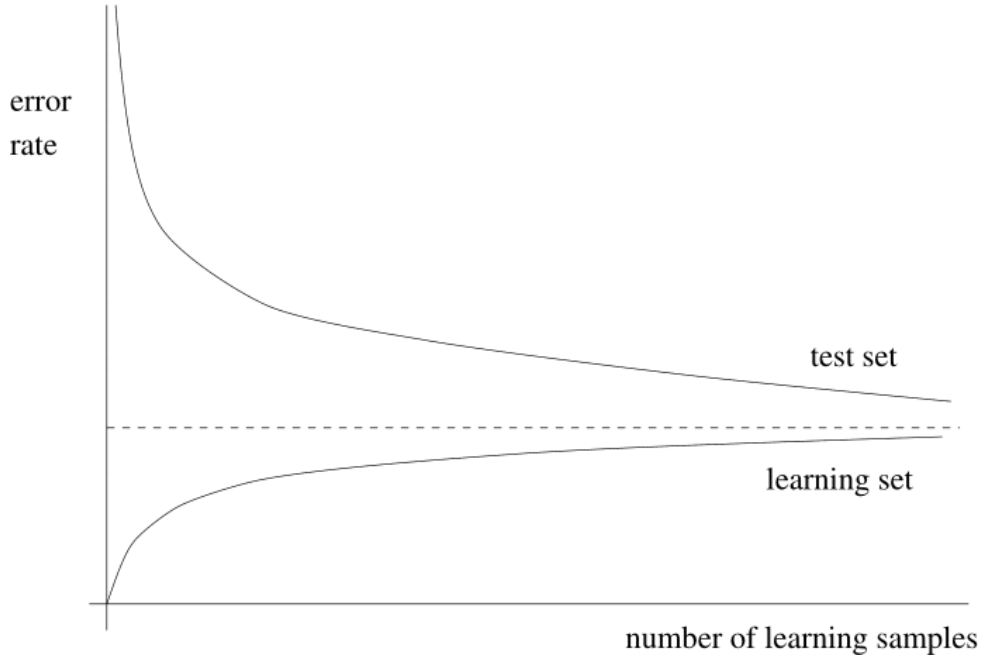


Figure 2.10.: "Effect of the learning set size on the error rate. The average error rate and the average test error rate as a function of the number of learning samples." [51, Figure 4.8]

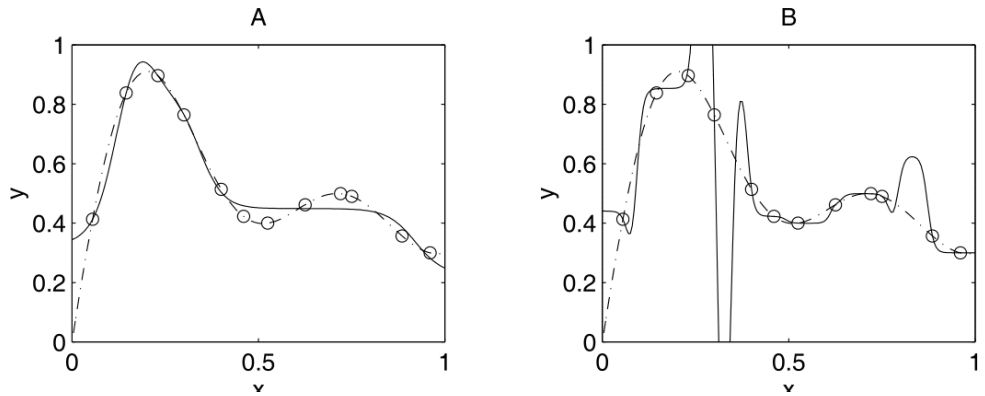


Figure A.3.: " Effect of the number of hidden units on the network performance. The dashed line gives the desired function, the circles denote the learning samples and the drawn line gives the approcimation by the network. 12 learning samples are used. a) 5 hidden untis. b) 20 hidden units." [51, Figure 4.9]

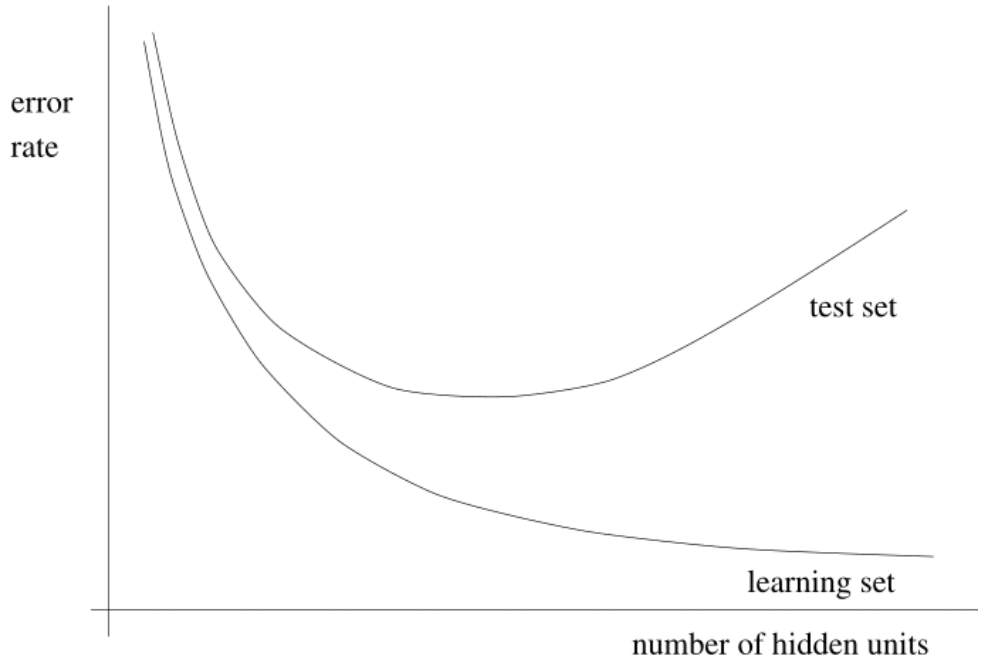


Figure A.4.: "The average learning error rate and the average test error rate as a function of the number of hidden units." [51, Figure 4.10]

A.2. Framework Scores (Kiviat Graphs)

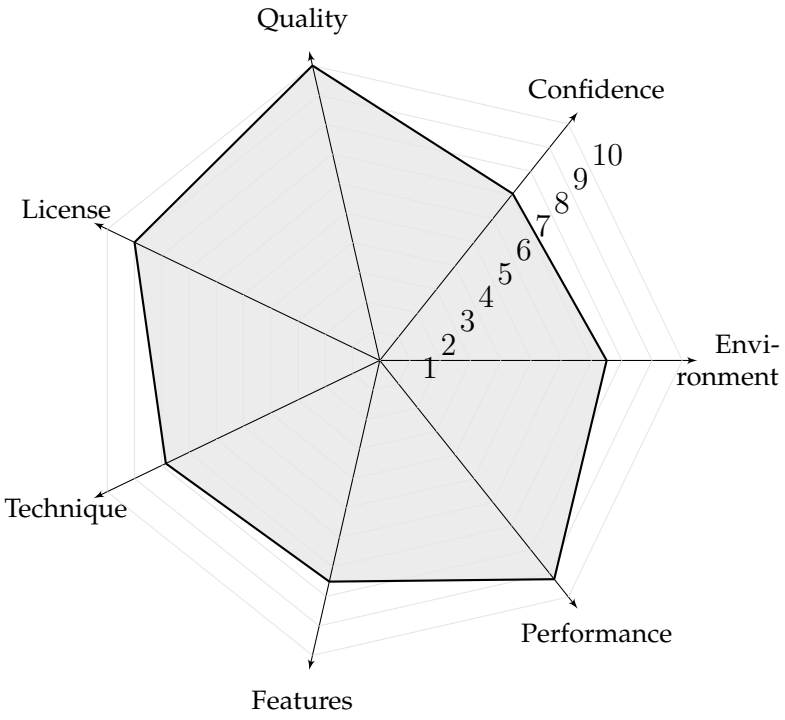


Figure A.5.: Caffe

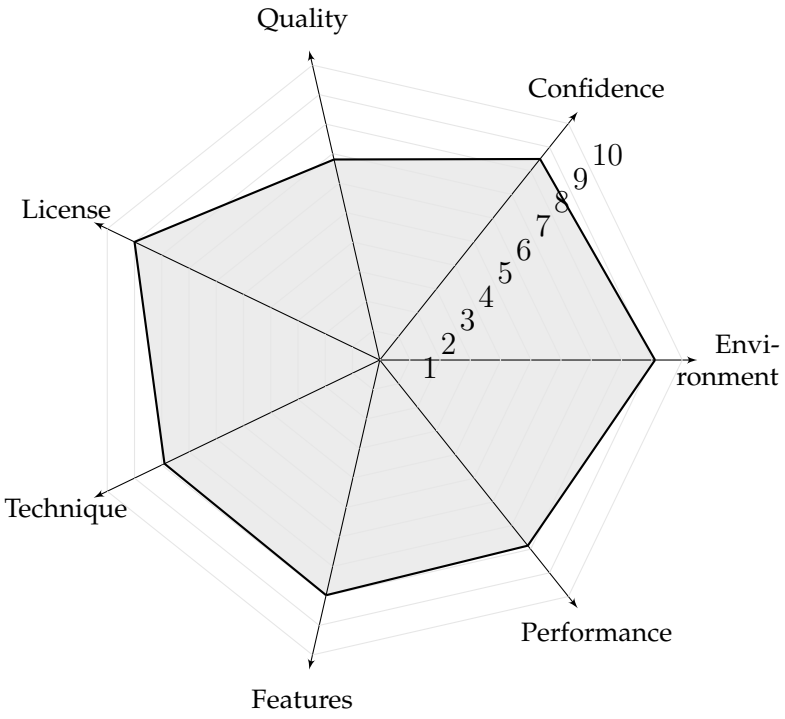


Figure A.6.: Caffe2

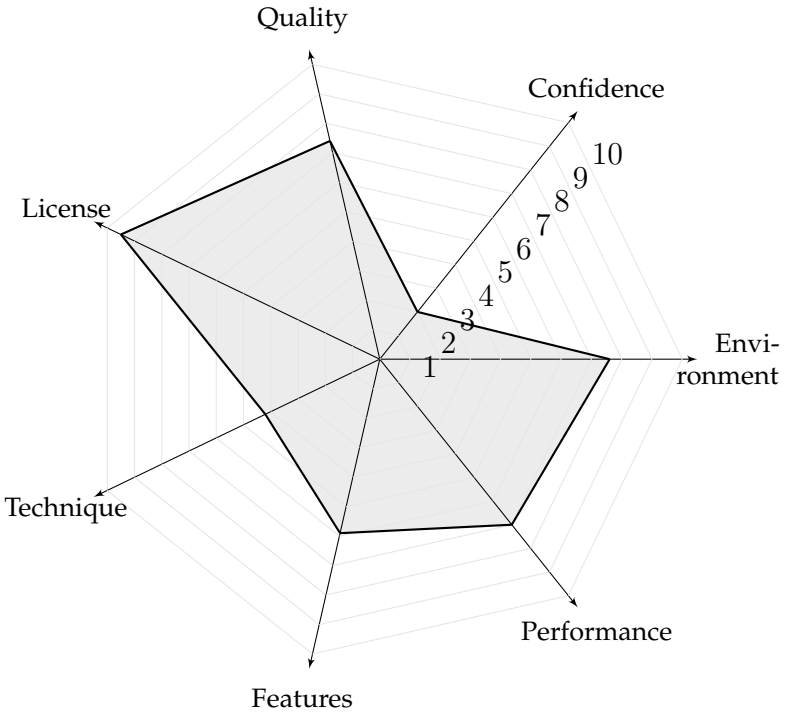


Figure A.7.: Keras

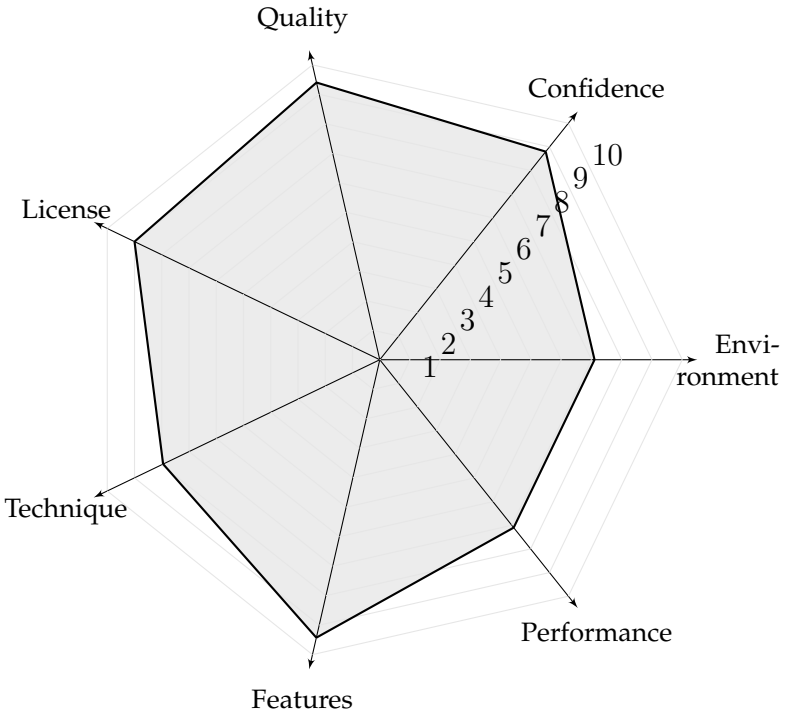


Figure A.8.: Scikit-Learn

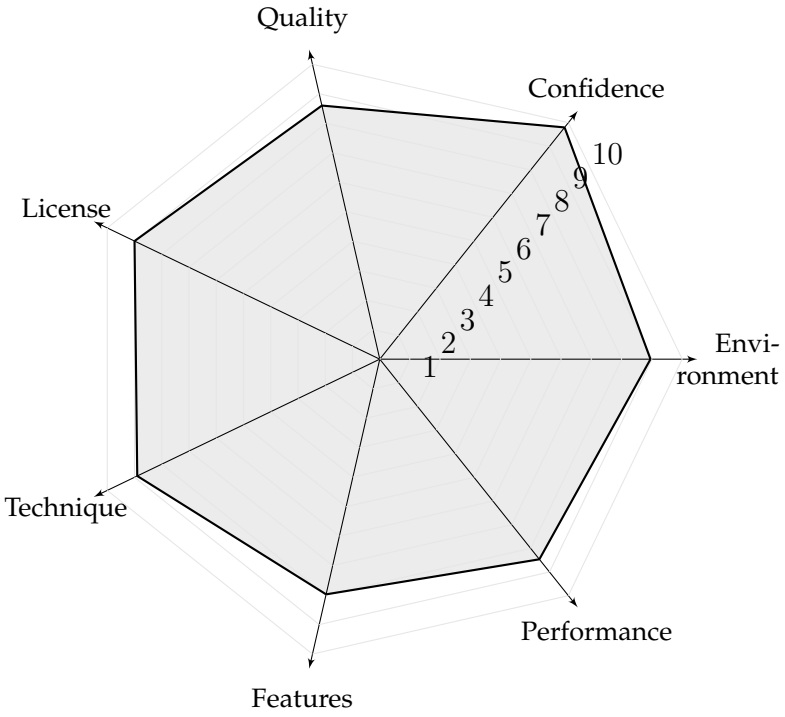


Figure A.9.: Tensorflow

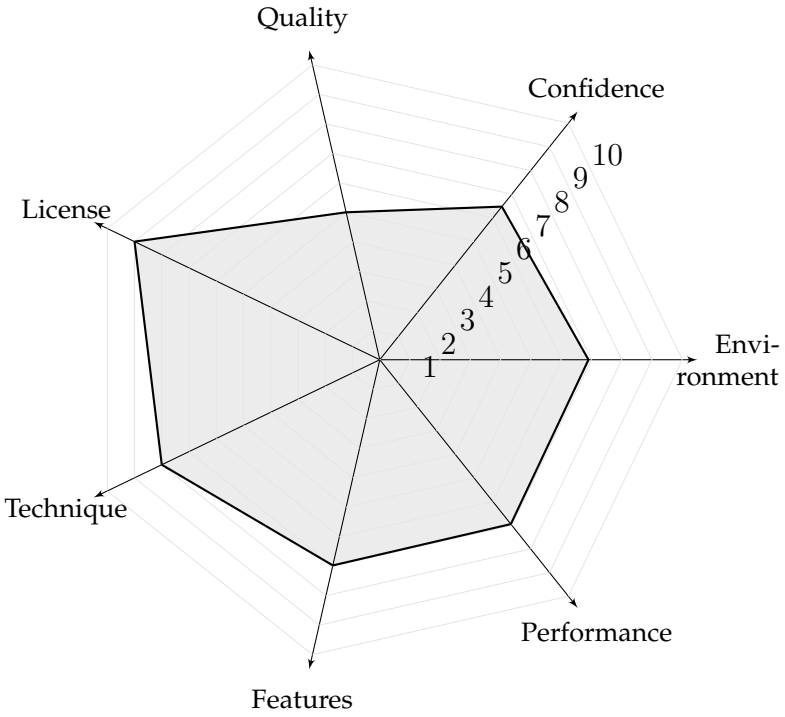


Figure A.10.: Theano

B. Additional coding examples

B.1. Neural network prototype

```
[{
  "name": "YM",
  "nnconfig": {
    "general": {
      "forecast_sec": 10,
      "learning_rate": 0.01,
      "training_epochs": 2,
      "batch_size": 10000,
      "display_step": 0.1,
      "parseDate": true,
      "calcNextSec": false,
      "predictPercChange": false,
      "debuglog": 1
    },
    "network": {
      "input": 8,
      "out": 1,
      "layer": [112, 50, 10],
      "activationfunction": {
      "layer": ["sigmoid"]
        "out": "",
    },
    "biases": [],
    "weights": []
    }
  },
  "trainsize": 0.7,
  "testsize": 0.3,
  "runsize": 0,
  "window": 0
  "savedModel": null,
  "savedModelPath": "/home/herget/ibm/BlueBull/app/result/YM.ckpt",
}, {
  "name": "IBM",
  ...
}]
```

Listing B.1: Example configuration file

```
try:
    date = dt.strptime(row[0]+' '+row[1], '%m/%d/%Y %H:%M:%S').replace(
        tzinfo=None)
    date_ms  = int(helper.unix_time_millis(date))
    if date_ms < fromd or fromd == 0:
        fromd = date_ms
    if date_ms > tod or tod == 0:
        tod = date_ms
    fn = dircode+date.strftime('%d-%m-%Y')+'.csv'
    if not fn == old:
        file.close()
        try:
            file = open(fn,'a')
        except FileNotFoundError:
            file = open(fn,'w')

    file.write(str(i)+','+str(date_ms)+','+row[2]+','+row[3]+'\n')
    old = fn
except:
    print("ERROR while seperating CSV: ("+",".join(row)+")")
```

Listing B.2: Implementation of a row based interpreation of the raw file

```
def sortFile(dircode,filename):
  pandasFrame = pd.read_csv(dircode+filename,
    sep=",",
    parse_dates=True,
    header=None,
    dtype={'id':'int','timestamp_ms':'int','price':'float','volume':'int'},
    names = ["id","timestamp_ms","price","volume"])
  pandasFrame = pandasFrame
    .groupby("timestamp_ms")
    .agg({'price': ['mean'],'volume': ['sum']})
  pandasFrame.sort_values(pandasFrame.columns[1])
  fname = os.path.splitext(os.path.basename(filename))[0]
  pandasFrame.to_hdf(dircode+fname+".h5", "table", mode='w')
  return pandasFrame.shape[0]
```

Listing B.3: Implementation of a row based interpreation of the raw file

```
def getTrainTestSets(df, start, end, config):
    if start == 0 and end == 0:
        batch_x = df.to_records(index=True)
    else:
        batch_x = df.iloc[start:end].to_records(index=True)
    batch_x_ret = []
    batch_y = []
    for i in range(0, len(batch_x)):
        x= batch_x[i]
        originaldate = dt.fromtimestamp(x[0]/1000)
        time = int(config["general"]["forecast_sec"]*1000+int(x[0]))
        found = False
        pprint = False
        p = 0
        if originaldate.day == dt.fromtimestamp(time/1000).day:
            if config["general"]["calcNextSec"]:
                while not found:
                    try:
```

```
                        if originaldate.day == dt.fromtimestamp(time/1000).
                            day:
                            forecast_value = df.loc[time]["price"]["mean"]
                        else:
                            forecast_value = batch_x[len(batch_x)-1][1]
                        found = True
                    except:
                        p += 1
                        pprint = True
                        time += 1000
                        found = False
            elif i+1 < len(batch_x):
                try:
                    forecast_value = df.loc[time]["price"]["mean"]
                except:
                    k = 1
                    ktime = dt.fromtimestamp(batch_x[i+k][0]/1000)
                    dif = ktime-originaldate
                    while ((ktime-originaldate).total_seconds() < config["
                        general"]["forecast_sec"] and k+i+1 < len(batch_x))
                        :
                        k+=1
                        ktime = dt.fromtimestamp(batch_x[i+k][0]/1000)
                    if k+i < len(batch_x):
                        forecast_value = batch_x[i+k][1]
                    else:
                        forecast_value = batch_x[len(batch_x)-1][1]
            else:
                forecast_value = x[1]
        else:
            forecast_value = batch_x[len(batch_x)-1][1]
        if pprint and config["general"]["debuglog"] > 2:
            print("    Timestamp",time,"not found. Used a timestamp",p,"
                seconds later.")
        if config["general"]["parseDate"]:
            batch_x_ret.append([originaldate.year, originaldate.month,
                originaldate.day, originaldate.hour, originaldate.minute,
                originaldate.second, x[1], x[2]])
        else:
            batch_x_ret.append([x[0],x[1],x[2]])
        if config["general"]["predictPercChange"]:
            forecast_value = (forecast_value-x[1])/x[1]
        batch_y.append([forecast_value])
    return np.array(batch_x_ret), np.array(batch_y)
```

Listing B.4: Generating of sets for training, testing and running

```
def trainNN(x, y, config, trainset, testset, code, n):
    results = {}
    weigths = calcWeighs(config)
    biases = calcBiases(config)
    layers,prediction = multilayer_perceptron(x, config, weigths, biases)

    cost = tf.reduce_mean(tf.square(tf.subtract(prediction,y)))
    optimizer = tf.train.GradientDescentOptimizer(config["general"]["
        learning_rate"]).minimize(cost)
    saver = tf.train.Saver()
    init = tf.global_variables_initializer()

```

```
    tfconfig = tf.ConfigProto()
    tfconfig.gpu_options.allow_growth = True
    with tf.Session(config=tfconfig) as sess:
        f = 0
        sess.run(init)
        print("Training neural network")
        results["epochs"] = {}
        for epoch in range(config["general"]["training_epochs"]):
            avg_cost = 0.
            results["epochs"][str(epoch)] =  {}
            startepochtimer = timer()
            starttimer = timer()
            for file in trainset:
                i = 0
                f+=1
                traindata, forecastdata, debug = getSetsPandas(file, config
                    )
                if debug:
                    while i < traindata.shape[0]:
                        start = i
                        end = i+config["general"]["batch_size"]
                        starttimerBatches = timer()
                        batch_x,batch_y = getTrainTestSets(traindata,
                            forecastdata, start, end, config)
                        endtimerBatches = timer()
                        i += config["general"]["batch_size"]
                        if config["general"]["debuglog"] > 1:
                            print("0%  ",batch_x[0], batch_y[0])
                            print("50% ",batch_x[int(round(len(batch_x)
                                /2,0))], batch_y[int(round(len(batch_y)
                                /2,0))])
                            print("100%",batch_x[(len(batch_x)-1)], batch_y
                                [(len(batch_y)-1)])
                        _, c = sess.run([optimizer, cost], feed_dict={x:
                            batch_x, y: batch_y})
                        avg_cost += c / config["general"]["batch_size"]
                        if f%(round(len(trainset)*config["general"]["
                            display_step"],0))==0 or f == 1 or f == len(
                            trainset) or config["general"]["debuglog"] > 0:
                            print("  ",code,"("+str(round(endtimerBatches-
                                starttimerBatches,2))+"/"+str(round(timer()
                                -starttimer,2))+" secs) Trainingfile",f,"(
                                Epoch:",epoch+1,") from",len(trainset),"(",
                                i,"/",traindata.shape[0],")",c)
                            starttimer = timer()
            f = 0
            timerDelta = round(timer()-startepochtimer,2)
            results["epochs"][str(epoch)]["avg_cost"] = avg_cost
            results["epochs"][str(epoch)]["secs"] = timerDelta
            print(' ',code,' ('+str(timerDelta)+' secs) Epoch', epoch+1, '
                completed out of',config["general"]["training_epochs"],'avg
                 loss:',avg_cost,"\n\n")
        if not config["general"]["predictPercChange"]:
            correct_pred = tf.less(tf.abs(tf.subtract(prediction, y)), 5)
        else:
            correct_pred = tf.equal(tf.divide(prediction, tf.abs(prediction
                )),tf.divide(y, tf.abs(y)))
```

```
54              correct_pred_full = tf.less(tf.abs(tf.subtract(prediction, y)),
                     0.0001)
55              accuracy_full = tf.reduce_mean(tf.cast(correct_pred_full,tf.
                    float32))
56              ac_full_mean = []
57          accuracy = tf.reduce_mean(tf.cast(correct_pred,tf.float32))
58          print("Prepare testfiles")
59          f= 0
60          accmean = []
61          starttimer = timer()
62          for file in testset:
63              f+=1
64              testdata, forecastdata, debug = getSetsPandas(file, config)
65              if debug:
66                  test_x,test_y = getTrainTestSets(testdata, forecastdata, 0,
                         0, config)
67                  ac = accuracy.eval({x:test_x, y:test_y})
68                  if config["general"]["predictPercChange"]:
69                      ac_full = accuracy_full.eval({x:test_x, y:test_y})
70                      ac_full_mean.append(ac_full)
71                  if config["general"]["debuglog"] > 2:
72                      print(test_x[0], test_y[0])
73                      print(test_x[int(round(len(test_x)/2,0))], test_y[int(
                            round(len(test_y)/2,0))])
74                      print(test_x[(len(test_x)-1)], test_y[(len(test_y)-1)])
75                      print('Accuracy (',f,'):',ac)
76                  accmean.append(ac)
77                  if f%round(len(trainset)*config["general"]["display_step"
                        ],0)==0 or f == 1 or config["general"]["debuglog"] > 0
                        and config["general"]["debuglog"] < 3:
78                      print("  ("+str(round(timer()-starttimer,2))+" secs)
                            Testingfile",f,"from",len(testset),"finished (",
                            file,")",ac)
79                      starttimer = timer()
80          acmean = 0.
81          for ac in accmean:
82              acmean += ac
83          acmean = acmean/len(accmean)
84          if config["general"]["predictPercChange"]:
85              acmean_full = 0.
86              for ac in ac_full_mean:
87                  acmean_full += ac
88              acmean_full = acmean_full/len(ac_full_mean)
89          if config["general"]["predictPercChange"]:
90              print(" The average accuracy is",acmean,"( Trend:",acmean_full,
                    ")")
91              results["accuracy_trend"] = acmean
92              results["accuracy_value"] = acmean_full
93          else:
94              print(" The average accuracy is",acmean)
95              results["accuracy"] = acmean
96          saveFile = path.join(here, 'result/'+str(code)+".ckpt")
97          saver.save(sess, saveFile)
98          print("Model is saved in",saveFile)
99          results["timestamp"] = helper.unix_time_millis(dt.now())
100         return saveFile, results, n
```

Listing B.5: Generating sets for training, testing and running

```
def doSet(filen, secs, calcNextSec, dirfname):
    df = pd.read_hdf(filen, "table", mode='r')
    del df["volume"]
    batch_x = df.to_records(index=True)
    batch_x_ret = []
    batch_y = []
    for i in range(0, len(batch_x)):
        x= batch_x[i]
        originaldate = dt.fromtimestamp(x[0]/1000)
        time = int(secs*1000+int(x[0]))
        found = False
        pprint = False
        p = 0
        if originaldate.day == dt.fromtimestamp(time/1000).day:
            if calcNextSec:
                while not found:
                    try:
                        if originaldate.day == dt.fromtimestamp(time/1000).
                            day:
                            forecast_value = df.loc[time]["price"]["mean"]
                        else:
                            forecast_value = batch_x[len(batch_x)-1][1]
                        found = True
                    except:
                        p += 1
                        pprint = True
                        time += 1000
                        found = False
            elif i+1 < len(batch_x):
                try:
                    forecast_value = df.loc[time]["price"]["mean"]
                except:
                    k = 1
                    ktime = dt.fromtimestamp(batch_x[i+k][0]/1000)
                    dif = ktime-originaldate
                    while ((ktime-originaldate).total_seconds() < secs and
                        k+i+1 < len(batch_x)):
                        k+=1
                        ktime = dt.fromtimestamp(batch_x[i+k][0]/1000)
                    if k+i < len(batch_x):
                        forecast_value = batch_x[i+k][1]
                    else:
                        forecast_value = batch_x[len(batch_x)-1][1]
            else:
                forecast_value = x[1]
        else:
            forecast_value = batch_x[len(batch_x)-1][1]
        batch_y.append([x[0], forecast_value])
    pfy = pd.DataFrame(batch_y)
    pfy = pfy.set_index([0])
    pfy.to_hdf(dirfname, "forecast", mode='w')

def doSingle(code, sec):
    pstart = timer()
    i=0
    dircode = path.join(here, 'data/data-'+code+'/')
    dircodesecs = path.join(here, 'data/data-'+code+'/sets-'+str(sec)+'/')
    statusfilepath = dircode+"status.json"
```

```
57      statusfilepathSecs = dircodesecs+"status.json"
58      try:
59          os.makedirs(dircodesecs)
60      except IOError:
61          print("  Folder ("+dircodesecs+") already exists")
62      if os.path.isfile(statusfilepath):
63          with open(statusfilepath) as data_file:
64              status = json.load(data_file)
65          if os.path.isfile(statusfilepathSecs):
66              with open(statusfilepathSecs) as data_file:
67                  statusSecs = json.load(data_file)
68          else:
69              statusSecs = {}
70              statusSecs["status"] = False
71              helper.writeStatusFileSecs(statusfilepathSecs, "false")
72          if status['status'] and status['sorted'] and not statusSecs["status
                "]:
73              helper.deleteFilesInFolder(dircodesecs, '.h5')
74              directory = os.fsencode(dircode)
75              initfiles = len(os.listdir(directory))-1
76              for file in os.listdir(directory):
77                  filename = os.fsdecode(file)
78                  if filename.endswith(".h5"):
79                      # print(os.path.join(directory, filename))
80                      doSet(str(dircode+filename), sec, False, str(
                            dircodesecs+filename))
81                      i+=1
82                      if i%100 == 0:
83                          pend = timer()
84                          print("  Prepared "+str(i)+"/"+str(initfiles)+"
                                files for",code,"(",sec,") ("+str(round(pend-
                                pstart,2))+" secs)")
85                          pstart = timer()
86                      continue
87              helper.writeStatusFileSecs(statusfilepathSecs, "true")
88          elif not status['status'] or not status['sorted']:
89              print("ERROR data not finished yet")
90          else:
91              print("  Set(",code,":",sec,") already prepared and ready to
                    use")
92      else:
93          print("ERROR no status file ("+statusfilepath+")")
94      return dircodesecs
95
96  def do(codes):
97      secs = {}
98      for rcode in codes:
99          secs[rcode["name"]] = []
100     for rcode in codes:
101         secs[rcode["name"]].append(rcode["nnconfig"]["general"]["
                forecast_sec"])
102     for rcode in codes:
103         secs[rcode["name"]] = sorted(set(secs[rcode["name"]]))
104         for sec in secs[rcode["name"]]:
105             doSingle(rcode["name"], sec)
106     print(secs)
```

Listing B.6: Outsourced generating sets for correct prediction values to local storage

B.2. Influencer gathering

```
{
  "scoringweights": {
    "watson": {
      "emotion": {
        "anger": [
          [0.2, -10]
        ],
        "disgust": [
          [0.2, -0.2]
        ],
        "fear": [
          [0.2, -15]
        ],
        "joy": [
          [0.2, 50]
        ],
        "sadness": [
          [0.2, -25]
        ]
      },
      "social": {
        "agreeableness_big5": [
          [0.5, -0.1],
          [0.75, 0.2]
        ],
        "conscientiousness_big5": [
          [0.5, 20],
          [0.75, -10]
        ],
        "emotional_range_big5": [
          [0.5, 0.1],
          [0.75, -10]
        ],
        "extraversion_big5": [
          [0.5, 0.15],
          [0.75, -10]
        ],
        "openness_big5": [
          [0.5, -5],
          [0.75, 0.25]
        ]
      },
      "language": {
        "analytical": [
          [0.5, 30]
        ],
        "confident": [
          [0.5, 0.25]
        ],
        "tentative": [
          [0.5, -15]
        ]
      }
    }
  },
  "catweights": {
```

```
    "watson": 1
  }
}
```

Listing B.7: Complete *scoring.json* example