DIPLOMARBEIT

Volatility forecasting in finance

ausgeführt am

Institut für
Stochastik und Wirtschaftsmathematik
TU Wien

unter der Anleitung von

Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Gerhold

durch

David Sonnbichler
Matrikelnummer: 01526963
[redacted]

Wien, am 17.07.2021

Kurzfassung

Aufgrund ihrer Bedeutung für Handels- und Hedgingstrategien ist die Volatilitätsprognose seit mehr als 40 Jahren ein aktives Forschungsgebiet. Die anspruchsvolle Aufgabe hat in letzter Zeit mit der erfolgreichen Implementierung von künstlichen neuronalen Netzwerken wieder an Bedeutung gewonnen, welche oft bessere Ergebnisse liefern als viele traditionelle ökonometrische Prognosemodelle. In dieser Arbeit werden Volatilitätsprognose eines long short-term memory (LSTM) Netzwerks mit zwei statistischen Benchmarkmodellen verglichen. Der Versuchsaufbau wurde so konzipiert, dass er verschiedene Szenarien abdeckt. Er enthält Prognosen für Kombinationen verschiedener Aktienindizes, Prognosehorizonte sowie verschiedene Volatilitätsschätzer als Zielgrößen für die Prognose. In den meisten Fällen konnte das LSTM Netzwerk einen niedrigeren quadratischen Fehler als die beiden Benchmark-Modelle GARCH(1,1) und die naive Random-Walk-Vorhersage erreichen. Die Ergebnisse stehen im Einklang mit bestehenden Studien zur Volatilitätsvorhersage und zeigen, dass für die untersuchten Aktienindizes ein LSTM Netzwerk in einer Vielzahl von Prognoseszenarien im Vergleich zu den gewählten Benchmarkmodellen zumindest wettbewerbsfähig und oft überlegen ist. Zusätzlich wurde für den S&P 500 Index die Performance LSTM Netzwerken mit zwei und drei hidden layers untersucht. Gegenüber dem LSTM Netzwerk mit einem hidden layer konnte keine klare Verbesserung festgestellt werden.

Abstract

Due to its importance for trading and hedging strategies, volatility prediction has been an active area of research for more than 40 years now. The challenging task has recently gained more traction again with the successful implementation of artificial neural network approaches, yielding better results than many traditional econometric forecasting models. In this thesis volatility predictions a long short-term memory neural network are compared to two statistical benchmark models. The experimental setup was designed to cover a range of different scenarios. It contains forecasts for combinations of different stock indices, forecasting horizons as well as different volatility estimators as target variables for the forecast. In the majority of cases the long short-term memory network outperformed the two benchmark models GARCH(1,1) and random walk method prediction. Results are in line with existing studies on volatility prediction and show that for the stock indices examined a long short-term memory network is at least competitive, and often superior, to the chosen benchmark models in a wide range of forecasting scenarios. Additionally for the S&P 500 Index the performance of long short-term memory models with two and three hidden layers was examined. No clear improvement could be found over the single hidden layer network.

Contents

1. Introduction

Accurate forecasts of stock market volatility can be of immense value for investors and risk managers. A good guess about a stocks expected deviation from its current price can ease the choice of trading and hedging strategies. Plenty of models have been proposed to forecast stock price time series volatility but it still remains a challenging task with no clear path how to approach it. The main issues in volatility forecasting are the difference of stock market behavior in different time periods but also the well observed phenomenon of volatility clustering, where periods of higher volatility are followed by periods of lower volatility and vice versa.

In this thesis we will tackle the task of volatility forecasting with an artificial neural network well suited for time series problems. A single layer Long Short-Term Memory neural network will be tested against a historical volatility model and a GARCH model under various circumstances. Moreover the single layer network will be tested against a two- and a three layer network to compare predictive capabilities.

The aim of this thesis is twofold. Firstly it differs from many 'proof of concept' works which show that in one specific situation good predictions can be achieved. In the following experiment different combinations of time horizons, volatility estimators and stock indices are combined when comparing the artificial neural network to two different benchmark models. This allows for more general conclusions to be drawn from results than other more specialized experiments. Secondly it might give an interested practitioner an idea how much can be gained by establishing an artificial neural network for volatility predictions in their respective environment.

Firstly chapter (2) contains a literature review, providing an overview about both the historical development and recent findings on models designed for volatility forecasting. The following chapters (3) and (4) will introduce commonly used volatility estimators as well as historical and econometric forecasting models. In chapter (5) theory and different architectures of neural networks, recurrent neural networks and long short term memory neural networks are summarized.

The main part of this thesis starts with chapter (6) introducing the experimental setup. A long short term memory model is tested against a historical volatility and a GARCH model for different stock indices, forecasting horizons and volatility estimators. The stock indices of choice are the standard & poor 500, DAX Performance Index and SSE Composite

Index. Time frames are mid- to short-term forecast horizons with 2,5,10 and 20 business days. For each model, stock index and time frame forecasts are obtained for three different volatility estimators. One calculated by continuously compounded squared returns, one range based estimator and five minute sampled realized volatility.

Results are summarized in chapter (7). In most cases the long short-term memory network could outperform the other two benchmark models. Especially compared to the GARCH(1,1) model the long short-term memory networks results show a clear improvement in forecasting capabilities. For the S&P 500 Index the single hidden layer LSTM network was compared to a two- and a three hidden layer LSTM network. The three LSTM networks performed quite similar in terms of mean squared error across all volatility estimators and time horizons.

Finally in chapter (8) it is discussed to what extend the result of this experiment can be generalized, what questions are still open and how additional studies could be constructed to gather more information.

2. Literature review

Volatility forecasting has been one of the most active area of research in econometrics for decades now. Stock price time series are auto-correlated and non stationary which poses problems in the modeling and forecasting volatility. Initially intuitive approaches dominated, such as assuming today's volatility as best estimate for the one tomorrow. This class of models that is based on (weighted) linear combinations of previous volatility observations is nowadays referred to as historical volatility models. Despite their simplicity they still perform better than some of their modern, more elaborate competitors. This emphasizes how difficult it is to accurately model and forecast stock market volatility.

Theoretically heavier models came up with Engles [10] work on ARCH models in 1980. Stock returns are modeled as a stochastic process with drift and a stochastic component, consisting of white noise scaled by the volatility process. A generalization by Bollersev [6] and the resulting generalized ARCH (GARCH) model has been widely used ever since. Many branches of GARCH models have been developed, among others Nonlinear Asymmetric GARCH and GJR-GARCH, exponential GARCH or Threshold GARCH. There are numerous studies comparing their forecasting performance. For an extensive study of 330 different models see Hansen [19] or for a more recent comparison on the much more volatile Bitcoin price see Katsiampa [24].

One obstacle already arises when questioning how volatility is measured. Since volatility is latent, even ex-post, different ways of estimating it emerged. Traditionally this has mostly been done with a 'close-to-close' estimator, comparing the closing prices of each day, yielding a measure of variability. In 1980 Parkinson [32] first argued that an estimator based on the highest and lowest daily values is more meaningful than the close-to-close estimator. In the following years a whole class of extreme value estimators emerged based on the same idea, but with different underlying assumptions about drift and jumps processes. Most notable are the Garman-Klaas [15], Rogers-Satchel [37] and Yang-Zhang [40] estimators. The most recent developments could be achieved due to the easier availability of high frequency data. Summing up intraday returns of short intervals during a day leads to realized variance measures. Realized variance has the desirable property that as the interval grid size goes towards zero, the realized variance equals the integrated variance, see Anderson [1] for example. Practical problems with microstructure noise with very short interval estimators once again led to several different classes of solutions.

Returning to the task of volatility forecasting, different deep learning and machine learning approaches gained traction in recent years. Their flexibility in recognizing complicated patterns and non-linear developments which can otherwise be hard to fully capture in an econometrical model make them an interesting tool for volatility forecasting. As for the choice of a neural network's architecture, the most common one is the class of recurrent neural networks. They conceptually fit the time series forecasting problem best since time series inputs can be added in chronological order, allowing the model to detect time-dependent patterns. Among successful implementations are a Jordan neural network in Arneric et al. [3] or Stefani et al. [39] in which several machine learning methods outperformed a GARCH(1,1) model. Even though there are options, recently most research has proven long short term memory (LSTM) neural networks developed by Hochreiter and Schmidhuber [21] most successful. Recently in a comparative experiment Bucci [7] showed that a LSTM and the closely related GRU model outperformed other neural networks like traditional feedforward networks, Elman neural networks or the aforementioned Jordan neural networks.

After the decision on a general neural network architecture type is made there are choices concerning parameters, hyperparameters and smaller neural network architecture choices. One approach gaining traction is to create hybrid models, combining a statistical model with an artificial neural network. Recent examples can be found on the precious metal market are Kristjanpoller and Hernández [25] and Hu et al. [23] or Maciel et al. [30] on financial volatility modeling. These hybrid models usually incorporate a statistical model's results an additional input to all relevant and available data from daily stock price movements. In Fernandes et al. [12] exogenous macroeconomic variables are included to improve forecasting accuracy. Another interesting approach is incorporating recent news and online searches into predictions. Sardelich and Manandhar [38] or Liu et al. [28] obtain good results by including sentiment analysis as input.

3. Volatility

This chapter is loosely based on 'A Practical Guide to Forecasting - Financial Market Volatility' (2005) by Ser-Huang Poon [35]. On several instances more recent studies are added to take current theoretic developments as well as new studies into account.

3.1. Introduction

Anticipating a stocks return is a key task for traders and risk managers and of no less importance is the degree of variation one has to expect over a certain time frame, since this is directly connected to the spread of possible outcomes. This degree of variation or spread of possible outcomes is referred to as volatility in finance. Volatility is of interest for every participant of the market since investment choices are heavily influenced by how much risk is connected to them. Therefore it is obvious that a precise volatility forecast is valuable whenever a tradable asset is involved.

Every forecast requires a underlying model it is based on. In mathematics an uncertain development such as an asset on the stock market will be modeled as a random variable. Computing volatility is straightforward in the case of a model where the stock price is given by discrete random variable. The task grows in effort substantially when using a more realistic continuous time model. Among the obstacles one encounters trying to forecast volatility is non-stationary and autocorrelated. The term volatility clustering refers to the empirical observation that periods of high volatility are followed by low volatility periods and vice versa.

In general volatility can only be calculated over a certain time frame, as there is no such thing as the degree of variation at one specific point in time. Daily, weekly, monthly and yearly volatility are all important quantities in finance. The volatility calculated by all previous observations is called unconditional volatility. Unconditional volatility is only of limited use for forecasts since it might not reflect the current situation well enough due to the non-stationarity and autocorrelation of financial time series data. Usually one is more interested in the conditional, time dependent volatility over a certain time frame.

Assets are usually modeled a continuous random variables and data is only available for discrete points in time volatility of previous time periods can not exactly be measured, it rather has to be estimated. The following chapters introduce the most common volatility estimators and discuss their advantages and disadvantages from both a theoretical and a more practical point of view.

3.2. Squared returns estimators

We will start with an intuitive derivation of the historically most important and still commonly used squared returns estimator. With daily data easily available online, a simple approach is to treat the calculate the variance of daily closing prices p_t at day t. Its average daily variance over a $T-t$ time frame from the known sample can then be simply computed as

$$\sigma^2_{r^2,t,T} = \frac{1}{T-1}\sum_{j=t}^{T}(r_j-\mu)^2, \tag{3.1}$$

where the continuously compounded daily return r is calculated by $r_t = ln(p_t/p_{t-1})$ and μ is the average return over the day $T-t$ day period. To arrive at the volatility we can always just take the square root. Often a further simplification is to let the mean μ be 0. In Figlewski [13] it is noted that this often even yields improved forecasting performance. The daily variance for day t is then simply obtained by

$$\sigma^2_{r^2,t} = r_t^2 \tag{3.2}$$

The volatility estimate obtained from equation (3.2) is an unbiased estimator when the price process p_t follows a geometrical Brownian Motion with no drift. Through

$$dln(p_t) = \sigma dB_t$$

close-to-close returns are calculated by $r_t = ln(p_t) - ln(p_{t-1}) = ln(p_t/p_{t-1})$. Since $r \sim N(0,\sigma^2)$ we have $\mathbb{E}[r^2] = \sigma^2$ and with that the squared continuously compounded returns are an estimate for the variance.

This is an intuitive approach but it does come with some drawbacks. Even though (3.1) is an unbiased estimator, as shown in for example Lopez [29] it is also quite noisy. A clear shortcoming is that since (3.1) does not take into account any intraday data, it can lack valuable information. A scenario with high intraday fluctuations but where closing prices randomly are close in price could lead to a false sense of security, since the squared returns estimator would suggest a calm stock price development.

3.3. Range based estimators

To circumvent this issue volatility estimators emerged which take daily extremes in the form og highest and lowest daily prices into account. The first of this class of range based estimator was developed by Parkinson in 1980 [32]. We follow the derivation [32] and let H_t and L_t be the highest and lowest price of any given day t, and O_t and C_t daily opening and closing prices. Similar to $r_t = ln(p_t/p_{t-1})$ we can calculate $h_t = ln(H_t/O_t)$ and $l_t = ln(L_t/O_t)$, describing the daily returns for the highest and lowest daily values reached. Additionally $d_t = h_t - l_t$ equals the difference daily highest and lowest returns. Let $P(x)$ be the probability that $d \leq x$. The probability was derived by Feller in 1951 [11].

$$P(x) = \sum_{n=1}^{\infty}(-1)^{n+1}n\left(erfc\left(\frac{(n+1)x}{\sqrt{2}\sigma}\right) - 2erfc\left(\frac{nx}{\sqrt{2}\sigma}\right) + erfc\left(\frac{(n-1)x}{\sqrt{2}\sigma}\right)\right)$$

where $erf(z) = \frac{2}{\pi}\int_0^z e^{-t^2}dt$ is the error function and the complementary error function is defined as $erfc(z) = 1 - erfc(z)$. The error function is a transformation of the cummulative distribution $\Phi(x)$ of the standard normal distribution with the relationship that $erf(z) = 2\Phi(x\sqrt{2}) - 1$. Parkinson calculated that for real $p \geq 1$ the expectation of d^p can be written as

$$\mathbb{E}[d^p] = \frac{4}{\sqrt{\pi}}\Gamma\left(\frac{p+1}{2}\right)\left(1 - \frac{4}{2^p}\right)\zeta(p-1)(2\sigma^2)^{p/2} \tag{3.3}$$

where $\Gamma(z)$ is the gamma function and $\zeta(z)$ is the Riemann Zeta function. In the case of $p = 2$ equation (3.3) simplifies to

$$\mathbb{E}[d^2] = 4ln(2)\sigma^2$$

this leads to the first range based volatility estimator, which is given by

$$\sigma^2_{Park,t} = \frac{(ln(H_t) - ln(L_t))^2}{4ln(2)} \tag{3.4}$$

Following Bollen and Inder [5] for the following range based estimators the assumption stands that the intraday price process follows a geometric Brownian motion. In the following years various other range based estimators appeared. Garman and Klass [15] argued for the superiority of an estimator which incorporates both open and close as well as high and low of the respective day. The standing assumption here is that the price process follows a geometric Brownian motion. Garman and Klass assume there is no drift, according to this they optimize their parameters and arrive at the Garman-Klass estimator

$$\sigma^2_{GK,t} = \frac{1}{2}\left(ln\left(\frac{H_t}{L_t}\right)\right)^2 - 0.39\left(ln\left(\frac{p_t}{p_{t-1}}\right)\right)^2 \tag{3.5}$$

Roger and Satchell [36] further developed the Garman Klass estimator by allowing for a drift. In 2000 Yang and Zhang [40] proposed a volatility approximation which takes overnight volatility jumps into account.

3.4. Realized variance measures

The most recent developments in volatility estimation are thanks to the availability of high frequency data. The idea is quite intuitive, arguing that the sum of measured intraday volatility converges against the integrated volatility with vanishing interval size. For the following derivation and discussion we will follow Chapter 1, 1.3.3 from Ser-Huang Poon [35].

We start with the assumption that the asset price p_t satisfies the following equation:

$$dp_t = \sigma_t dW_t, \tag{3.6}$$

yielding a continuous time martingale thanks to the characteristics of the Brownian motion, σ_t being a time dependent scaling variable which models the volatility. Now the integrated volatility

$$\int_t^{t+1} \sigma_t^2 ds$$

equals the variance of $r_{t+1} = p_{t+1} - p_t$. Here it is important to note that we cannot directly observe σ_t, since it is inferred by equation (3.6) and scales dW_t continuously. We circumvent this problem by first starting to measure the returns at discrete points $r_{m,t} = p_t - p_{t-1/m}$ with a given sampling frequency m and defining the realized volatility measure as

$$RV_{t+1} = \sum_{j=1,..,m} r^2_{m,t+j/m} \tag{3.7}$$

We chose $r_{m,t+j/m}$ in such a way that as m approaches infinity we cover the same t to $t+1$ interval is we already did with the integrated volatility. Through the theory of quadratic variation one can show that

$$\lim_{x\to\infty}\left(\int_t^{t+1} \sigma_t^2 ds - \sum_{j=1,..,m} r^2_{m,t+j/m}\right) = 0, \text{in probability} \tag{3.8}$$

for a detailed discussion on the theoretical argument see Andersen et al. [1].

Concretely for a 7 hour 30 minutes trading day the RV5 realized variance will be calculated with 90 five minute intervals prices as

$$RV_{t+1} = \sum_{j=1,..,90} r^2_{90,t+j/90} = \sum_{j=1,..,90} (p_{t+j/90} - p_{t+j/90-1/90})^2$$

In a world without microstructure noise the realized volatility would be the maximum likelihood estimator and efficient. However this noise exists, induces autocorrelation on the observed returns and for high sampling frequencies the realized volatility estimates will be biased [[26]]. To strike a balance between using as much high frequency data as possible, but also avoid the bias from microstructure noise, most often frequencies between one and fifteen minutes are chosen.

Based on RV measures many other estimators have been proposed. Estimator have been developed with the goal of removing the bias induced from microstructure noise, with the same idea first-order autocorrelation-adjusted RV tries to deal with the issue. Combinations of higher and lower frequency RV exist, also with the goal of reducing the effect of microstructure noise. Different realized kernel measures exist, and also realized range based variance has been implemented. Another whole subclass of realized measures included jump components to the various estimators. Over 400 of these different estimators have been tested against the standard five minute sampled realized variance, and it was found that the latter baseline model was hard to outperform [26]. For a deeper dive into the subject of realized measures either of Andersen et al. (2006) [2], Barndorff-Nielsen and Shephard (2007) [4] or recently Floros et al 2020 [14] provide good starting points.

3.5. Choice of volatility estimators

Three different volatility estimators were chosen for the experimental part of the thesis. The squared returns estimator, the Parkinson estimator and five minute sampled realized volatility will be the three target variables for different forecasting models and time horizons. This covers the widely used squared returns estimator as well as the classes of range based estimators and realized measures. The following chapter will give an explanation why out of the numerous options in each class those three were decided on.

As this should also be a guide of practical value the squared returns estimator (3.4) with average return $\mu = 0$ is included. Despite its drawbacks it is commonly used due to its simplicity to implement, and also ease to explain and understand.

As a representative of the family of range based estimators the Parkinson volatility is chosen. One reason for the choice is that Petneházi and Gáll [34] show similar predictive behavior for a LSTM network for all range based estimators is shown. They tried to forecast the up and down movements for each previously mentioned range based volatility estimators with a LSTM network and concluded that not much of a difference could be observed. They did find that it was easier to predict than the squared returns estimator though. This is in line with the findings of the experiment in this thesis, where the Parkinson estimate forecasts consistently yielded a lower MSE than the forecasts for the squared returns estimator.

With this result in mind the decision on the Parkinson volatility is based on its simplicity of implementation. More elaborate estimates can get quite complicated and are therefore also more unlikely to be practically implemented. If one is interested in implementing a more advanced ranged based volatility estimator, than a LSTM networks forecasting results should be comparable to the ones of the Parkinson estimator.

Finally we include one of the realized measures into our selection of volatility estimates. As discussed above, many different realized measures have been suggested. In Liu et al.[26] it is concluded that out of 400 tested estimators it is difficult to significantly outperform the RV5 measure. In their broad experimental setup they compare the performance of around 400 estimators on 31 different assets that are either exchange rates, interest rates, equities or indices. The forecasting horizons are short to mid term and range from one to 50 business days. Depending on the benchmark model and error measurement five minute realized variance was either found to be the best, or only very slightly outperformed, by other realized measures. Those results, as well as the ease of availability of daily calculated data from the [Oxford-Man Institute's realized library, published by Heber, Gerd, Lunde, Shephard & Sheppard (2009)] led to the choice of RV5 as a third volatility estimator.

3.6. Scaling

Often we will be interested in longer horizons than a single day for measuring or forecasting volatility. For a n day period starting at day t the n day volatility estimate $\sigma_{t,t+n}$ is given by

$$\sigma_{t,t+n} = \sqrt{\sum_{i=t}^{t+n} \sigma_i^2} \tag{3.9}$$

where $\sigma_i,\ i \in \{t, ..., t+n\}$ is the daily volatility calculated by any of the volatility estimators in section (3.1).

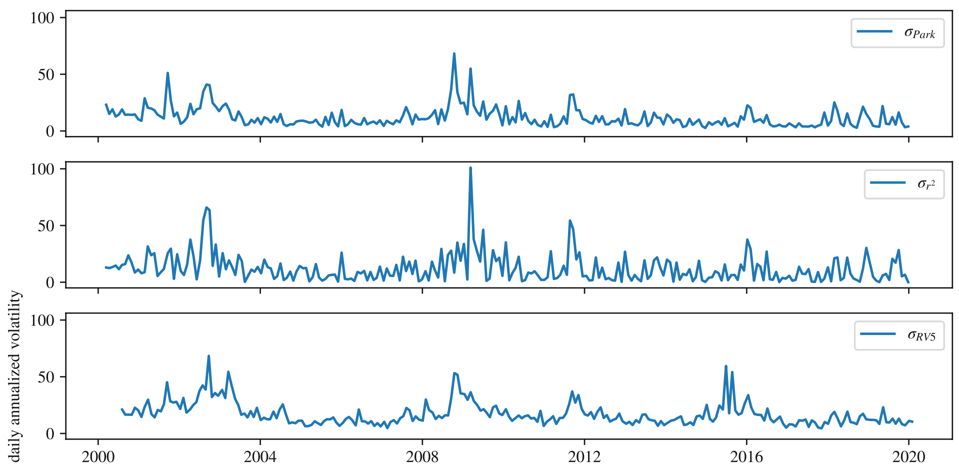

Figure 3.1.: Daily annualized volatility estimators of the DAX performance index

3.7. Comparing volatility estimators

In figure (3.7) the three selected volatility estimators daily continuously compounded squared returns, Parkinson volatility and realized volatility (5 minute sampled) are compared for the DAX performance index. The patterns are typical for these volatility estimators. 5 minute sampled realized volatility is scarce on outliers and extreme highs and lows while the squared returns estimator portrays a more volatile environment. The Parkinson estimator is situated somewhere in between.

The differences can be explained by considering what data is included in each estimator. While the realized measure has plenty if information available with one data point every five minutes during trading hours, the squared returns estimator only includes a single observation per day, leading to a noisier estimator. A scenario with high intraday price fluctuations but similar closing prices on two consecutive days will be depicted as a calm day for the squared daily returns estimator, but recognized as a high volatility day with extreme value estimators and realized volatility measures.

4. Historical and statistical models

In this chapter traditional approaches of forecasting volatility are summarized and based on literature and previous studies their performance is evaluated. The choice of statistical benchmark models which will test the artificial neural networks performance falls on a random walk and GARCH model due to simplicity, common practical use and their prevalence as a benchmark in many other studies of this kind.

To forecast the squared returns volatility estimator historically three classes of models have been most significant in practice and research. Those are historical volatility models, the family of ARCH models and stochastic volatility models. Each of them will be introduced in the following sections. Besides those three classes in recent times other econometrical models have proven more successful to accurately forecast high frequency volatility estimators such as five minute sampled realized variance. At the end of this chapter the HAR-RV model as well as the ARFIMA model will be introduced.

4.1. Historical volatility models

The class of historical volatility models mostly provides simple and intuitive forecasts based on some recent past observations. One approach is to forecast the next time steps volatility by the one of the current time step, e.g. we always use today's volatility as a forecast for the one tomorrow.

$$\hat{\sigma}_{t+1} = \sigma_t \tag{4.1}$$

A moving average model calculates the average volatility of the last n periods and will use this as a prediction for the next periods forecast.

$$\hat{\sigma}_{t+1} = \frac{\sigma_t + \sigma_{t-1} + ... + \sigma_{t-n}}{n} \tag{4.2}$$

Since in practice it is observed that today's volatility depends more heavily on yesterdays volatility then on the one of days further back, this can be taken into account as well by adding scaling parameters to each observation. The moving average model can be altered by adding exponential weights to the past n observations such that more weight is given to more recent observations. The resulting exponentially weighted moving average model is already more elaborate then the previous ones, also requiring to be fit on previous in sample observations to choose the exponential smoothing parameters. Forecasts are obtained by

$$\hat{\sigma}_{t+1} = \frac{\sum_{i=1}^{n} \beta^i \sigma_{t-i-1}}{\sum_{i=1}^{n} \beta^i} \tag{4.3}$$

4.2. GARCH models

The second class of models contains various variants of the AutoRegressive Conditional Heteroscedasticity (ARCH), first developed by Engle [10]. ARCH models assume that returns are distributed through

$$r_t = \mathbb{E}[r_t] + \epsilon_t, \tag{4.4}$$

where

$$\epsilon_t = \sigma_t z_t, \tag{4.5}$$

with a white noise process z_t, scaled by the time dependent volatility σ_t. While z_t usually is normally distributed, other distributions as the t-student distribution or the skewed t-student distribution have also been tested.

Now in the ARCH(q) model the variance is given as

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2, \tag{4.6}$$

where for all $i \in \{1, ..., q\}$ the scaling parameters $\alpha_i \geq 0$. Here we can see the difference to the class of historical volatility models. While the next time steps volatility is still deterministic, it depends on the previous observations of the process ϵ_t, not σ_t itself. Now we are left with parameters α_i which need to be fit to the time series. The most common way of fitting is using the maximum likelihood method, even though other methods as Markov Chain Monte Carlo have been tried as well more recently.

The ARCH model was generalized by Bollerslev [6], also considering the previous values of the variance itself, not only its past squared residuals. The resulting GARCH(p,q) model has the following variance structure:

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{p} \alpha_i \epsilon_{t-i}^2 + \sum_{j=1}^{q} \beta_j \sigma_{t-j}^2 \tag{4.7}$$

where $p \geq 0$, $q > 0$, $\alpha_0 > 0$, for $i \in \{1, ..., q\}$ the parameters $\alpha_i \geq 0$ and for $j \in \{1, ..., p\}$ the parameters $\beta_j \geq 0$. This allows for the choice of $p = 0$, reducing the model to an

ARCH(q) one again. Similar to the ARCH model we fit the GARCH model using the maximum likelihood method, also discussed in Bollerslev [6].

Since then numerous iterations and potential improvements have been proposed. The exponential GARCH (EGARCH) model, developed by Nelson [31], models the variance differently. Here the variance is based on a function of lagged innovations introducing an asymmetry in how the model reacts to positive or negative developments. Asymmetry is also considered in NAGARCH or GJR-GARCH models. For an extensive survey on different GARCH models and their predictive ability see Hansen and Lunde [19] where 330 GARCH type models are compared. The experiment tested whether the standard GARCH(1,1) model would be outperformed by any of the other 330 GARCH-type models. The models were compared under six different error measurements for the IBM daily returns and the DM - \$ exchange rate. As target variable a realized variance measure was chosen, as it most closely resembles the latent conditional variance. It was concluded that for the exchange rate no model was clearly superior to the GARCH(1,1) model. For the daily IBM returns the GARCH(1,1) model was outperformed by several other models that accommodate for a leverage effect.

For the GARCH class of families there is a choice to be made between different approaches. Suppose we want to forecast a n day period. Let p_t be the daily closing price and $r_{t,n} = p_t - p_{t-n}$ the n day period return. One way of obtaining a forecast for the next period would be to feed the model with a series $\{r_{i,n}, r_{i+n,n,...,r_{t-n,n},r_{t,n}}\}$ of previous n day returns. Now our standard GARCH model will produce a deterministic one period forecast for the variance of the next n day period. One issue with this approach is that the model loses information by disregarding daily returns between the days with which the n day returns are measured. For a monthly forecast for example, it could be that the stock price is extremely volatile during the month but close to its monthly starting price again at the end. Since the monthly return would be close to zero the model handles it like a low volatility period.

The other approach is formulated using daily returns and working with the assumption that $\mathbb{E}[\epsilon_{t+n}^2] = \sigma_{t+n}^2$, to generate recursive forecasts. Assuming we are currently at time step t and want to use a GARCH(1,1) model to forecast the daily n day ahead volatility σ_{t+n} the recursion would have the following form. Let $n \geq 2$, for the one day forecast we have

$$\sigma_{t+1}^2 = \alpha_0 + \alpha\epsilon_t^2 + \beta\sigma_t^2 \tag{4.8}$$

through which we than can expand the daily forecasts to an arbitrary horizon

$$\begin{aligned}\sigma_{t+n}^2 &= \alpha_0 + \alpha\sigma_{t+n-1}^2 + \beta\sigma_{t+n-1}^2 \\ &= \alpha_0 + (\alpha+\beta)\sigma_{t+n-1}^2\end{aligned} \tag{4.9}$$

With future daily volatility forecasts for $\sigma_t, \sigma_{t+1}, ..., \sigma_{t+n-1}, \sigma_{t+n}$ we will now proceed as we did with the interval calculation of volatility in chapter (2) and calculate the n day period volatility forecast by

$$\sigma_{t,n} = \sqrt{\sum_{i=1}^{n} \sigma_{t+i}^2}. \tag{4.10}$$

This can easily be generalized to a GARCH(p,q) model by taking more lagged observations into account and exchanging the previous day variances in equation (4.8) and (4.9) with the sums of previous variances.

While the daily forecasts get noisier with growing n, the forecasts generated from daily returns still show improved results over the ones from n day returns, where a simple one step forecast is generated. This is generally observed in shorter forecasting periods, for longer horizons, starting from several months on, the one step model is often preferred [13]. For a monthly horizon Ñíguez [18] shows that daily multi step forecasts provide the most accurate forecasts. As we only consider short to medium length horizons up to 20 business days we will stick to recursively generated daily forecasts as a benchmark.

4.3. Stochastic volatility models

For this section the stock price process will follow a Brownian motion with drift. As the name suggests, all Stochastic Volatility (SV) models have in common that a stocks volatility is once again modeled as a stochastic process. A general representation of the class of stochastic volatility models for some functions $\alpha_{V,t}$ and $\beta_{V,t}$, stock price S_t, drift μ, volatility process V_t and Brownian motions dW_t^S and dW_t^V is given by

$$\begin{aligned}dS_t &= \mu S_t dt + \sqrt{V} S_t dW_t^S \\ dV_t &= \alpha_{V,t} dt + \beta_{V,t} dW_t^V\end{aligned} \tag{4.11}$$

One early representative os the SV class of models is the Heston [20] model. The continuous time model follows the stochastic equations

$$\begin{aligned} dS &= \mu S_t dt + \sqrt{V} S_t dW_t^S \\ dV &= \kappa(\theta - V_t)dt + \xi\sqrt{V_t} dW_t^V \end{aligned} \tag{4.12}$$

with constants μ, $\kappa \geq 0$, ξ, $\theta > 0$. While ξ scales the stochastic volatility, θ models the long variance. For an extending time horizon we observe that $\lim_{t\to\inf} \mathbb{E}[V_t] = \theta$. The mean reversion rate κ determines the speed of convergence to θ.

Once the framework is established how stock price and volatility stochastically behave, one has to implement a method for forecasting the next periods volatility. In comparison to the historical volatility models or GARCH class of models it is not as straightforward on how to make a prediction for any period. Recently the approach that has yielded the best result is the Monte Carlo Markov Chain method. It describes a class of algorithms for sampling from a probability distribution.

4.4. Other models

For a better overview on the field of volatility forecasting in this section other non-neural network models that are used for volatility forecasting are reviewed.

With the rise of realized volatility measures as a standard approach for estimating volatility the question arises how to forecast it most accurately. The widely used GARCH models often show a rather poor performance. This can be explained due to a GARCH models 'blindness' to any intraday developments. To combat this problem heterogeneous AutoRegressive models of realized volatility (HAR-RV) and the Autoregressive fractionally integrated moving average (ARFIMA) model have proven especially successful. For this reason this chapter will be concluded with those two models.

HAR-RV models

This subsection follows Corsi (2008)[8].

The HAR-RV arose from the observation, that traders behave differently dependent on their time horizons. While day traders looking for profit will often respond immediately to market changes, while an insurance company with a very long term investment will not be as concerned by today's smaller market movements. In accordance with ones goals daily, weekly, or monthly re-balancing and repositioning of ones portfolio are considered to be standard intervals. Long term traders are not as concerned with short term volatility spikes, but short term traders are influenced by long term volatility outlooks and changes. This leads to the following structural dependencies in the model. See Corsi [8] for a more detailed discussion of the argument.

Every component of the model follows an AR(1) structure. For every but the longest time frame its development is also affected by the expectation of the next longer time frame. The resulting cascading structure can compactly be summarized by substitution. Let RV_t^d be the daily realized volatility introduced in section (3.4). Following the HAR-RV model the realized volatility can be computed by

$$RV_{t+1}^d = \alpha + \beta^d RV_t^d + \beta^w RV_t^w + \beta^m RV_t^m + \omega_{t+1} \tag{4.13}$$

Where RV^m, RV^w and RV^d are the daily, weekly and monthly realized volatility. Parameters are usually fit using the ordinary least square method, minimizing the sum of errors.

AR(FI)MA models

Autoregressive moving average models (ARMA) are a common tool to forecast time series data. For time series data S_t the ARMA(p,q) model can be described by the equation

$$(1 - \sum_{i=1}^{p} \alpha_i B^i) S_t = (1 + \sum_{i=1}^{q} \beta_i B^i) \epsilon_t$$

where $B^i S_t = S_{t-i}$ is the backshift operator, ϵ_t a white noise process. The parameters α_i and β_i are fit to previous observations of the time series. ARMA models can model and forecast stationary stochastic processes, but are not able to deal with non-stationary data. Since volatility is non-stationary a simple ARMA model will not be sufficient for forecasting.

The ARIMA model originates from a AutoRegressive moving average model, but through differencing it is possible to eliminate non stationarity of the mean. The AutoRegressive fractionally integrated moving average (ARFIMA) model once again generalizes the ARIMA model by allowing for non-integer parameter values. The ARIMA(p,q,d) and ARFIMA(p,q,d) models can be written as

$$(1 - \sum_{i=1}^{p} \alpha_i B^i)(1 - B)^d S_t = (1 + \sum_{i=1}^{q} \beta_i B^i) \epsilon_t$$

For $d = 0$ the extra term $(1 - B)^d$ equals one and with that the ARIMA model is simplified to an ARMA model again while for $d = 1$ we have $(1 - B) S_t = S_t - S_{t-1}$, the one time step difference of the time series. An ARFIMA model additionally allows for fractional inputs for d. In that case the term $(1 - B)^d$ is defined as

$$(1-B)^d = \sum_{k=0}^{\infty} \binom{d}{k} (-B)^k = \sum_{k=0}^{\infty} \frac{\prod_{a=0}^{k-1}(d-a)(-B)^k}{k!}$$

4.5. Model selection

To measure our LSTM's performance we choose the random walk model (3.4) and a GARCH(1,1) model (4.7) with aggregated daily forecasts for multiple day horizons. The choice was based on the following considerations:

- Forecasting performance
- Common use in research and literature
- Common use in practice
- Ease of implementation

The forecasting performance is rather obvious, since there is little to be gained in showing a neural networks superiority over models known for poor performance. For comparison purposes it is convenient when the same models as in similar research papers are used. Often it is hard to keep an overview over new developments when too many different target variables, error metrics and models are used.

The last two points in the list are heavily intertwined. The implementation of a theoretically advanced and therefore often simulation and computation heavy model will take plenty of effort and as complexity grows, so does room for errors. In comparison to more straightforward models they can also become difficult to explain when the results differ significantly from more intuitive models.

With this in mind the decision was made to include one model of the class of historical volatility models as well as a GARCH model. Here we disregard the class of stochastic volatility models since the models are complex, rely on a heavy body of theory and the implementation can be laborious. In addition to that there is a scarcity on studies examining how well SV models perform for different forecasting problems. In Poon [35], chapter 7, an overview about studies examining SV forecasting performance can be found. The results range from promising to quite poor performance. In most cases traditional methods of volatility forecasting seem to at least be on par with SV predictions.

For the historical volatility models we will proceed to use the random walk prediction resulting from the random walk model as a simple benchmark. Despite its simplicity it can be surprisingly hard to outperform it significantly, emphasizing how difficult of a task volatility forecasting is in general.

The second model we will test against the LSTM is the standard GARCH(1,1) model. It sees plenty of practical use and if so, only gets slightly outperformed by its more elaborate counterparts. Moreover since many research papers test their respective models against a simple GARCH implementation it helps with comparability.

As for scaling the models to multiple periods for the random walk prediction we will always use last periods volatility as a forecast for the next period. Here $\sigma_{t,n}$ represents the volatility estimate of choice at time t over the previous n business days. It is determined by one of the volatility estimates discussed in chapter (3). The simple estimate for next period is obtained by

$$\hat{\sigma}_{t+n,n} = \sigma_{t,n}. \tag{4.14}$$

The GARCH model will be fed with daily data and multi day forecasts will be recursively generated as described in section (4.2).

5. Neural networks

Chapter (5) will start with a brief introduction to artificial neural networks (ANN), traditional feedforward neural networks (FNN) and recurrent neural networks (RNN). In the following sections there will be particular focus on long short term memory models (LSTM) since they show the best practical results for time series forecasting problems and will be implemented for the empirical part of the thesis.

Introductory example

Artificial neural networks recognize patterns through a process of trial and error, resembling a human way of learning. As a slightly unorthodox introductory example the game of chess will serve here. The reign of chess engines over human players is established for some time already. In 1995 the at the time world champion Garry Kasparov lost to the chess-playing computer 'Deep Blue', developed by IBM. Previous to that it was considered impossible for computers to beat strong human players since there are too many possible variations to simply brute-force ones way through every possible outcome. Deep Blue circumvented the calculation of so many different variations by introducing an evaluation function that estimates how good a position is if a move is made. The evaluation function is quite complex and consists of hundreds of impactful factors such as 'how protected is my king?', 'what pieces are there to capture', 'how well positioned are my pawns' and so on. With high quality chess games the chess computer was then trained to weight those parameters. Chess engines like the open source program Stockfish or Rybka refined this process with improving parameter choices, weighting and more efficient algorithms to determine which moves should be evaluated further.

This process resembles an econometric model such as an exponentially weighted moving average model to forecast volatility. First human thought is put into the question what parameters are important and then through some statistical approach the model is fit to data.

In 2018 the company DeepMind released AlphaZero, a chess engine based on deep neural networks. AlphaZero remarkably was able to outperform state of the art chess engines without any human expert knowledge or ever seeing another game played. In the beginning the chess engine only knows the rules of chess and starts from a blank starting point. It learns the game by playing itself repeatedly, starting with making completely random moves. It will then through many games against itself learn which moves more often than others lead to wins in certain positions. Through this flexible process of trial and error AlphaZero adapts a human like play style that relies on its 'intuition' from millions of previously played games. A strong human player will usually 'feel' which are the most promising moves in the position, before very selectively calculating a few critical lines.

This approach of learning is in essence also how the long short-term memory network will predict volatility. Instead of specifying how the previous n days of previous volatility, open or close prices and any other factor will determine future volatility, the network is fed with data and through training on the previous years it will come to its own conclusions how important each factor is.

Categorization of neural networks

A neural network can learn in different ways, which are useful for different situations. They can be split into the three groups of supervised learning, unsupervised learning and reinforcement learning.

The previous example of mastering the game of chess describes the process of reinforcement learning. An action has to be performed in a specified environment to maximize some kind of value. In the game of chess the action is to make a move in the current environment, which is the position on the board. Finally the value to maximize is the probability to win the game from the current position by making any legal move on the board.

Unsupervised learning is commonly used for tasks such as clustering (e.g. dividing objects into groups) or other tasks that require to find out something about the underlying data. Through unsupervised learning objects such as words, numbers or pictures can be examined and grouped together with other similar objects.

The most straightforward learning technique is supervised learning. For training inputs and already known correct outputs are given to the neural network. In training the network will learn how the inputs are connected to the output and use this knowledge for future forecasts. This is the technique used for volatility forecasting. The outputs are previous values of volatility estimators which can be calculated and are known. Inputs are usually the respective days stock data such as daily open, close, highest and lowest values, even though additional or less inputs are possible as well.

For supervised learning the learning process will be carried out as follows. At first random weights are assigned to input values, since no knowledge of any possible correlations between input and output exists at the start of training. The input values will then pass through multiple layers of mostly nonlinear functions, called neurons. These initially random weights are what enable the learning process. In training the results produced by the neural network are compared to the desired output and weights will be adjusted to lower some form of error measurement. For numerical values mean absolute error or mean squared error are popular choices. Through repeating this process many times the in-sample error will decrease further and further since the weights will be better and better adjusted to the patterns of the training set. This repetitive procedure allows to capture complex patterns and nonlinear pattern. Even a one hidden layer neural network is (with a sufficient amount of hidden neurons) a universal approximator, implying that with a sufficient number of hidden nodes a wide range of nonlinear functions can be approximated by the network.

This general process will be more thoroughly explained in the next sections for different neural network architectures.

5.1. Feedforward neural networks

FNNs are organized in layers. The information flow is best observed graphically as in figure (5.1). Input data is passed on through a number of functions until an output value is produced.

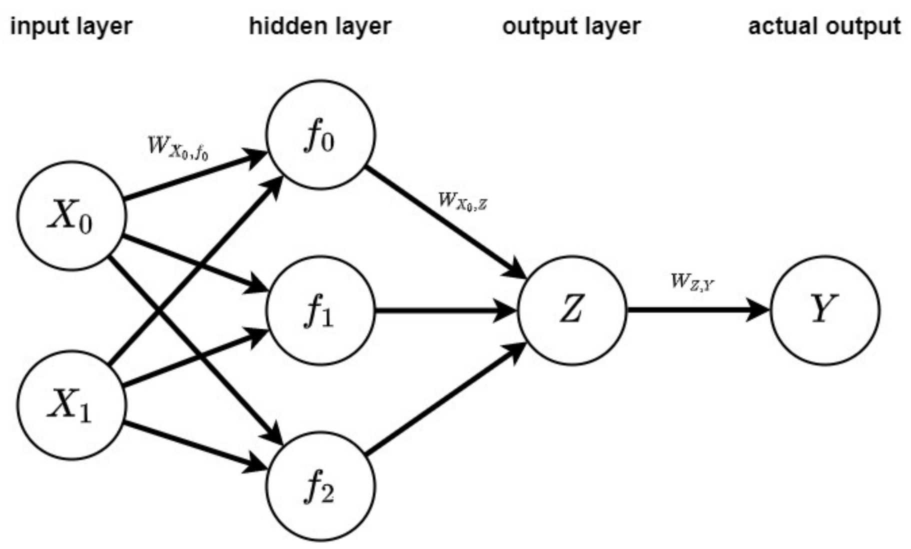

Figure 5.1.: Feedforward neural network with two inputs, one hidden layer with three nodes and one output.

The connected functions in the neural network are called neurons or nodes, hence the name neural network. Any neuron will first receive a sum of weighted inputs and then add a bias, a constant term. The weighted sum plus bias will then pass through the actual function defining the neuron. This so called activation function is usually nonlinear, allowing the ANN to capture complex structures. By no means the only but a common choice is the sigmoid function $\sigma(x) = 1/(1 + e^{-x})$. Other frequently used activation functions or the binary step function, tanh or rectified linear unit.

A hidden layers neurons output will then be calculated by

$$y = \sigma\left(b + \sum_{i=1}^{n} w_{i,j} x_i\right), \tag{5.1}$$

where x_i is the input from the input layer, $w_{i,j}$ the corresponding weights and a bias b.

To enable pattern recognition the neural network first runs through a training set where the desired output is known and available. After the neural network produces an output it is compared to this desired output. The predictions quality is assessed by computing some error measurement between the actual and desired output. Weights are then adjusted to lower the in-sample error after. Every such iteration is called an epoch. The most common method to reduce this error is a backpropagation algorithm in combination with a stochastic gradient descent algorithm. Often incorrectly only referred to as backpropagation [16]. This combinations of algorithms is the backbone of a many neural networks. It allows the neural network to efficiently train trough many epochs by using algorithms that only use a small fraction of the computational effort of less sophisticated algorithms for the same purpose. Due to its importance it will be explained in more detail in the following subsections.

The subsections 'Stochastic gradient descent' and 'Backpropagation' are based on Goodfellow et. al [16], where a detailed discussion can be found.

Stochastic gradient descent

For a given objective function $f(x)$ with $x \in \mathbb{R}^n, n \in \mathbb{N}$ the gradient descent algorithm finds input values to minimize the target function. By moving a small step into the opposite sign of the derivative the algorithm will end up close to a local minimum $f(x) = 0$. For $n > 1$ the gradient $\nabla f(x) = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n})$ will be calculated to find the minimum. For a given learning rate (in non machine learning literature also step size) ϵ, a new point

$$x' = x - \epsilon \delta f(x)$$

is proposed. The learning rate ϵ can be chosen in different ways, the easiest one being to pick a small constant value. This procedure can then be repeated until all partial derivatives are sufficiently close to 0, which implies that the local minimum of $f(x)$ is found. This algorithm is called gradient descent. One problem that occurs with the algorithm is that it can computationally grow very heavy. For large data sets and many input variables the training time can reach exhaustive lengths that make using the algorithm unattractive. Consider a supervised learning network with n observations and weight vector $x \in \mathbb{R}^n, n \in \mathbb{N}$. The loss function

$$L(x) = \frac{1}{n} \sum_{i=1}^{n} L_i(x)$$

calculates the average loss depending on the set of weights x. The loss functions is not further specified here, common choices are mean absolute error or mean squared error between the predicted outcome and the desired outcome. Applying the gradient descent algorithm thousands of times for data sets where n is in the millions will be extremely inefficient.

Stochastic gradient descend significantly improves computational cost by only minimizing the loss of a subset of observations $M \in 1, 2, ..., n$ where $|M| < n$. The subset will be drastically smaller than the number of total observations, usually under a couple hundred observations. The resulting estimator

$$L_m(x) = \frac{1}{|M|} \sum_{j \in M} L_j(x)$$

has the expected value $\mathbb{E}[L_m(x)] = L(x)$. By only using a small fraction of the computational power of the gradient descent algorithm the stochastic gradient descent algorithm can go through numerous iterations of smaller batches in the time it would take to go through one with all observations. Due to this is commonly used in machine learning applications instead of the standard gradient descend.

Backpropagation

While gradient descend algorithms are used to minimize the error function through its gradient values, backpropagation algorithms calculate the partial derivatives themselves. For ease of notation and better intuition the algorithm is exemplary explained for a vector of inputs and a single output. The same concept can be extended to matrix valued inputs and multi dimensional output.

A prerequisite is to define the term operation. An operation is a simple function in one or more variables. A simple example for an operation is the multiplication of $i, j \in \mathbb{R}$, yielding $i \cdot j = ij$. Importantly more complicated functions can be build combining multiple operations. The output of a hidden layer $y = \sigma\left(b + \sum_{i=1}^{n} w_{i,j} x_i\right)$ can also be described as a composition of operations. Each addition, multiplication but also the application of the

σ function is an operation.

Our aim is to calculate the gradient of a real valued function $f(x, y), x \in \mathbb{R}^m, y \in \mathbb{R}^n$ with respect to x. The vector y contains other inputs where the derivative is not required.

Suppose that moreover $g : \mathbb{R}^m \to \mathbb{R}^n, \mathbb{R}^n \to \mathbb{R}, y = g(x)$ and $z = f(g(x)) = f(y)$. The chain rule states that

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

or equivalently

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^\intercal \nabla_y z$$

where

$$\left(\frac{\partial y}{\partial x}\right) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

is the Jacobian Matrix. Recall that $z = f(g(x)) = f(y)$. For each simple function in the network the gradient can be calculated efficiently by the product of the Jacobian Matrix and the gradient $\nabla_y z$. The whole neural network can be viewed as a composition of operations, and for each of them this procedure is repeated until the first layer is reached.

The algorithm may be understand best by the simple example in figure (5.2), where the graphs and operations are displayed.

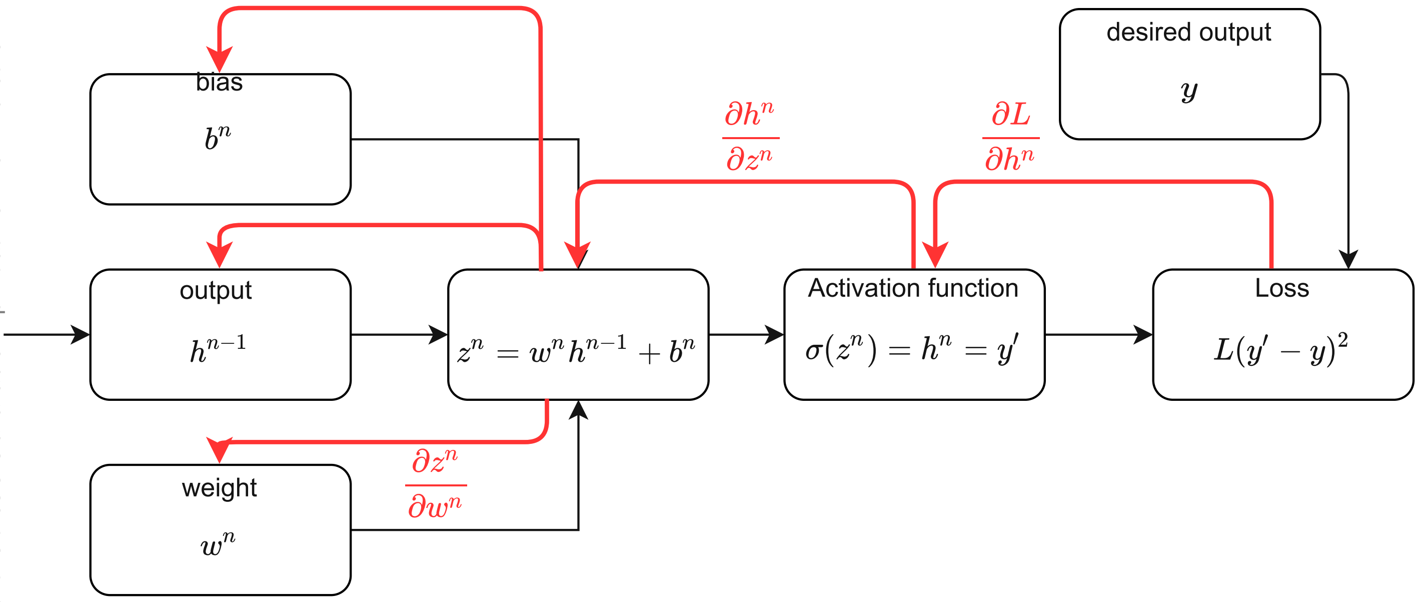
bias
b^n
output
h^{n-1}
weight
w^n
$z^n = w^n h^{n-1} + b^n$
Activation function
$\sigma(z^n) = h^n = y'$
Loss
$L(y' - y)^2$
desired output
y
$\frac{\partial h^n}{\partial z^n}$
$\frac{\partial L}{\partial h^n}$
$\frac{\partial z^n}{\partial w^n}$

Figure 5.2.: Backpropagation algorithm for a feedward neural network. The partial differential of the Loss function with respect to the weight w_n is calculated via the chain rule. It holds that $\frac{\partial L}{\partial w_n} = \frac{\partial L}{\partial h_n} \frac{\partial h_n}{\partial z_n} \frac{\partial z_n}{\partial w_n}$. The partial derivatives of the other weights are calculated in a similar manner, going back until hidden state h_t for $\frac{\partial L}{\partial w_t}$. Since the partial derivatives for $t < n$ can make use of the already calculated derivatives the backpropagation algorithm is light on computational cost.

Universal approximation theorem

In numerous variations it has been shown that a one hidden layer feedforward network as an approximator for a wide range of functions. Cybenko proves [9] the following variation of the universal approximation theorem for continuous functions: Let I_n denote $[0,1]^n$, $C[I_n]$ be the space of continuous functions over I_n and for every $f \in C[I_n]$ let $\|f\|$ denote the supremum norm. A sigmoidal function is defined as

$$\sigma(t) = \begin{cases} 1 & as\ t \to \infty \\ 0 & as\ t \to -\infty \end{cases}$$

Usually a sigmoid function will be monotonically increasing, but for this result here this assumption is not needed.

Now Cybenko [9] shows the following

Theorem. *Let σ be any continuous sigmoidal function. Let $x \in \mathbb{R}^n$ and fix the other variables $\gamma_j \in \mathbb{R}^n$, $\alpha_j \in \mathbb{R}$ and $\theta \in \mathbb{R}$.*

Then finite sums of the form

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(\gamma_j^T x + \theta_j) \tag{5.2}$$

are dense in $C[I_n]$.

This proves linear combinations as in equation (5.2) can approximate any continuous function on the n-dimensional unit cube $[0,1]^n$. The universal approximation theorem can also be generalized further to show that they can approximate any well behaved continuous function on $\mathbb{R}^n$, see for example Hornik (1991) [22].

5.2. Recurrent neural networks

Feedforward neural networks can suffer from their one dimensional architecture. As seen in figure (5.1) information only flows in one direction. This makes it unsuitable for non stationary time series problems like volatility prediction. For problems like this recurrent neural networks (RNNs) are frequently used. The term recurrent neural networks refers to neural network architecture that allow for sideways or backwards information flow in its hidden layers. A loop like structure has proven to work particularly well for time series problems. It can capture time dependent patterns, keeping previous events in mind and incorporating them into predictions.

In figure (5.2) a RNNs structure is outlined. Every time a new input x_t is added it passes through a neural network N. The neural network also receives its previous state as input, thus updating itself with every new input to take new developments into account. A simple implementation of the neural network N could be a *tanh* layer, mapping all values between -1 and 1.

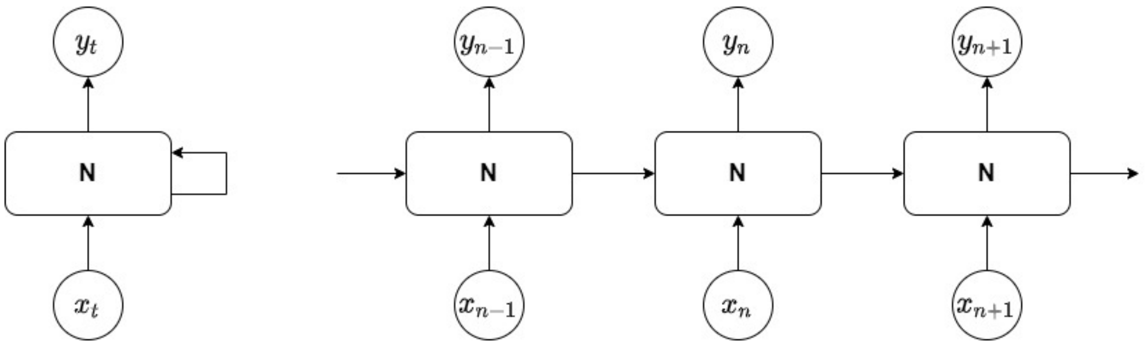

Figure 5.3.: Loop-like structure of a recurrent neural network

A simple RNN as in figure (5.2) with a *tanh* activation function may be described by the following equations

$$\begin{aligned} z_t &= b + W_H h_{t-1} + W_I x_t \\ h_t &= tanh(z_t) \\ y_t' &= c + W_O h_t, \end{aligned} \tag{5.3}$$

where b and c are bias vectors, W_H, W_I and W_O are weight matrices for the hidden state, input and output. The initial state h_0 has to be given to the model.

In training weights will be adjusted through a backpropagation with stochastic gradient descent algorithm minimizing the forecasting error for the training set. For a RNN this process is called backpropagation through time.

One issue that can arise in training is the vanishing gradient problem, where hidden states further back in time will decline in weight and therefore importance to forecasts. This is due to the gradient shrinking with each step back, when computing derivatives with respect to the weights The problem will be explained in a more thorough theoretical framework in the following section.

Vanishing gradient problem

Bengio et al. (1994) shows that for a simple RNN it is not possible to detect long-term dependencies for lengths 10 to 20 for gradient based optimization. To emphasize why the vanishing gradient is such a big issue in traditional RNNs the following example from language processing is often given. A sentence in a book starts as something like 'The temperature was low, James was ... '. When a RNN should predict the next word in a text a well trained network will probably make a sensible guess such as 'cold' or 'freezing' because the key words of temperature and low are still in recent memory, while 'James was' implies that a verb is likely to follow. If the sentence however is interrupted such as 'The temperature is low. 'Great', James thought, he had not properly prepared for today and now he was ... ' then the important context is already out of reach.

Depending on the specific problem statement the prove of the vanishing gradient problem will differ. Here following Pascanu et. al [33] it is shown for a simple recurrent network. First recall the structure of a RNN depicted in figure (5.2) and the backpropagation algorithm from subsection (5.1). Partial derivatives with respect to previous hidden states are calculated by the chain rule. Let the recurrent neural network have the following form

$$\begin{aligned} z_t &= W_H h_{t-1} + W_I x_t \\ h_t &= id(z_t) = z_t \\ y'_t &= h_t \end{aligned} \tag{5.4}$$

with h_0 specified, and W_H and W_I weight matrices and the identity as activation function. The output from the neural network is vector y' while desired output vector is denoted as y and $y, y' \in \mathbb{R}^n$. Assume an error function

$$L(y', y) = \frac{1}{n} \sum_{i=1}^{n} L_i(y', y) = \frac{1}{n} \sum_{i=1}^{n} (y'_i - y_i)^2 \tag{5.5}$$

The backpropagation through time algorithm will compute the gradient $\frac{\partial L(y',y)}{\partial W}$, where $W = (W_H, W_I)$. Pascanu [33] describes the problem as follows by rewriting the gradient to

$$\frac{\partial L_m(y', y)}{\partial W} = \sum_{k=1}^{m} \frac{\partial L_m(y', y)}{\partial h_m} \frac{\partial h_m}{\partial h_k} \frac{\partial^+ h_k}{\partial W} \tag{5.6}$$

where $\frac{\partial^+ h_k}{\partial W}$ describes the derivative of h_k with respect to W with h_{k-1} fixed with respect to W. Furthermore

$$\frac{\partial h_m}{\partial h_k} = \prod_{i=k}^{m} \frac{\partial h_i}{\partial h_{i-1}} \tag{5.7}$$

by design of the recurrent network. As already stated in section (5.1) the term $\frac{\partial h_i}{\partial h_{i-1}}$ can be expressed as Jocobian Matrix, and with that $\frac{\partial h_m}{\partial h_k}$ as the sum of Jacobian matrices.

Every summand $\frac{\partial L_m(y',y)}{\partial h_m} \frac{\partial h_m}{\partial h_k} \frac{\partial^+ h_k}{\partial W}$ of $\frac{\partial L_m(y',y)}{\partial W}$ is referred to as temporal contribution. The temporal contributions measures how at step k the weights W influence the error function at a later time m. Combined the sum equals the overall impact of the current weights on the error function. Without giving a clear definition of the term when m is much larger than k it is referred to as a long term contribution.

The spectral radius $\rho(A)$ of a matrix A is defined as the maximum value of the matrix absolute values of its eigenvalues. Pascanu et. al show that if $\rho(W_H) < 1$ it is sufficient for the gradient to vanish as m approaches infinity. In the same paper the result is also generalized to nonlinear activation functions.

5.3. Long Short Term Memory

First developed by Hochreiter and Schmidhuber [21] the long short-term memory (LSTM) network deal with the vanishing gradient problem, allowing information to persist. This is achieved by a persistent cell state. Input and forget gates carefully decide when information should be added or disregarded in the memory cell. With each new input they regulate which information is added and forgotten in the memory cell. This architecture allows long time dependencies to be detected, while in a recurrent neural network without memory cell events many iterations past will slowly get lost.

The following sections is loosely based on the excellent blog post [Colahs Blogpost]. For great illustrations, intuitive examples and a deeper dive on this topic it is highly recommended to read it.

Formally an LSTM model can be described through the following equations.

$$\begin{aligned} f_t &= \sigma(W_f[h_{t-1}, x_t] + b_f) \\ i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= tanh(W_C[h_{t-1}, x_t] + b_C) \\ C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \\ o_t &= \sigma(W_o[h_{t-1,x_t}] + b_o) \\ h_t &= o_t * tanh(C_t) \end{aligned} \tag{5.8}$$

with observations x_t, weight matrices W_f, W_i, W_C, W_o, biases b_f, b_i, b_C and b_o. The memory cells state is described by C_t, while h_t is the cells output at time t. With $*$ we denoate the element-wise product, σ a sigmoid function mapping the every input between 0 and 1. The vector $\tilde{C}_t$ contains candidate values which might be added to the cell state, this will be explained in more detail walking through the equations below.

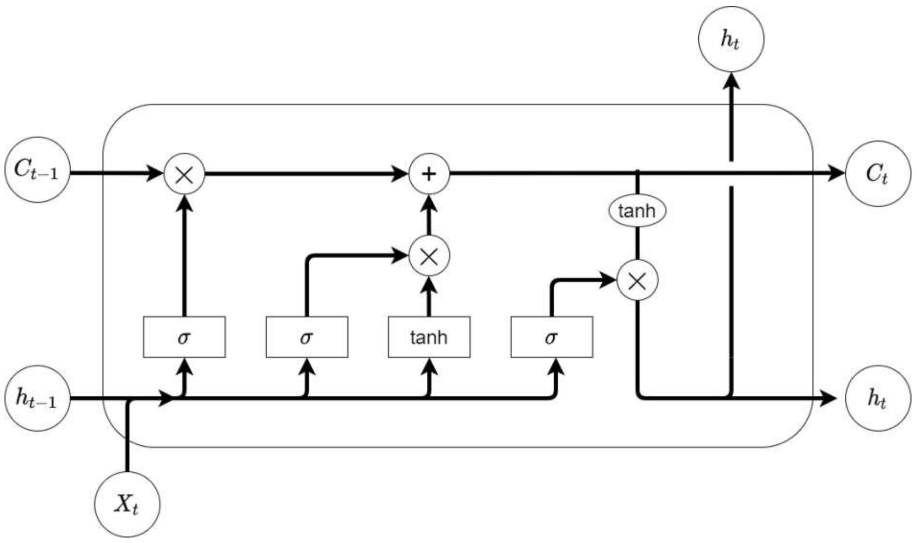

Figure 5.4.: Repeating module in a LSTM, visualizing how the cell state is updated with every new input and how the output is generated.

The equations describing the LSTM model are best understood having figure (5.3) in mind. For each new input x_t the following process is repeated. Assume that here we just passed time $t-1$ and will now add a new input at time t.

The key component of the LSTM model is the cell state C_t. It is updated with every new input through three different gates. First the forget gate decides which information from C_{t-1} is not necessary anymore. The first equation

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{5.9}$$

describes this process. The previous output, also called hidden state, h_{t-1} and new input x_t are multiplied by their respective weights and then pass through a sigmoid function, mapping all inputs between 0 and 1. A value of 0 implies 'forget everything' while a value of 1 implies that all information persists.

The next step is to add new information. A vector $\tilde{C}_t$ provides candidate values, suggesting possible additions to the cell state. The candidate vector fulfills the equation

$$\tilde{C}_t = tanh(W_C[h_{t-1}, x_t] + b_C). \tag{5.10}$$

The *tanh* function scales the output between -1 and 1. Not all of those values will be added to the cell state. An input gate i_t regulates what values are allowed to pass. It is constructed in similar manner to f_t, satisfying the equation

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i). \tag{5.11}$$

Note that the functions i_t and f_t will still not have identical output since their respective weight matrices differ. The process of adding information is compactly summarized by equation

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t. \tag{5.12}$$

The new cell state equals the previous one multiplied with the forget layer f_t plus certain new values which are suggested by the candidate vector $\tilde{C}_t$ and allowed to pass by the input gate i_t.

The last two equations

$$o_t = \sigma(W_o[h_{t-1,x_t}] + b_o) \tag{5.13}$$

and

$$h_t = o_t * tanh(C_t \tag{5.14}$$

determine the actual output h_t. The output gate o_t regulates which values of the memory cell will be used for determining the output. Information stored in the memory cell passes a *tanh* layer and is then combined with the output gate to create this iterations output and at the same time hidden state h_t for the next iteration.

5.4. LSTM variations

Based on the architecture in section (5.3) some variations of the LSTM architecture have been developed. Two LSTM variations that have already proven to yield good results in various circumstances is presented below.

Gated Recurrent Units (GRU)

A Gated Recurrent Unit (GRU) is closely related to LSTM networks. The difference is that the GRU lacks an output gate. In a GRU the output equals the hidden cell state at the time, the input and output gates are updated to an 'update' gate. Even thought the LSTM networks architecture is somehow simplified by taking away the output gate, in some experiments it has proven to provide superior forecasts. This tends to happen when datasets are smaller and less frequent, see Gruber and Jockisch (2020) [17]. Since daily or even intraday stock data is not scarce usually, it does not seem to be favorable for volatility predictions.

Peephole LSTM

The peephole variation of the LSTM network adds additional connections between the cell state and gate layers. In a standard LSTM the gate layers don't have any information about the cell state, but only about the previous output. Different peephole variations exist, not all gate layers have to gain information from the cell state, it is also possible to only include a single peephole somewhere. With this additional information for updating the cell state patterns might be detected more efficiently.

6. Data and experimental setup

In chapter (6) the data and experimental setup will be discussed. A LSTM network as described in chapter (5) is tested against two benchmark models. Those are the widely used GARCH(1,1) model and the simple random walk model. The experiment includes different time frames, different forecasting horizons and volatility estimators. The detailed choices of those, as well as error measurement of predictions and stock indices chosen for the comparison are explained in the sections below. Moreover data sources, preparation and implementation are explained.

6.1. Experimental setup - forecasting horizon and error measurement

Due to constraints in form of overall data availability and the need to split a dataset in training, validation and test set the time period for comparisons of different models is limited. The four year period from 01.01.2016 - 01.01.2020 allows for the biggest interval on which the LSTM can still be properly trained. Regarding the forecast horizons were we aim for short to medium length forecasts. In risk management, trading and option pricing for short time horizons with the next weeks ahead in mind are often required. In the experiment we conduct forecasts for 2,5,10 and 20 business days ahead, roughly covering monthly, weekly as well as two week and two business day forecasts.

For error measurement of predictions compared to the actual volatility estimates we follow the majority of studies and experiments on this topic and will stick to the common mean squared error (MSE). Let Y be the vector of actual values and $\hat{Y}$ be the predicted values, then the MSE is computed by

$$MSE_{Y,\hat{Y}} = \frac{1}{n}\sum_{i=1}^{n}(Y - \hat{Y})^2, \tag{6.1}$$

simply squaring and then averaging the errors of every predicted time step.

The results in 7 will be averaged across the 3 indices, notable deviations in the results of the 3 different indices will be mentioned below.

6.2. Experimental setup - stocks and data sources

The choice was made to consider mainly stocks and indices that are frequently traded, so that even though it might not be a safe bet to generalize the results to a broad range of financial assets, there still remains practical use in just the results of this single experiment. With this in mind the 3 indices chosen for the experiment are:

- Standard & poor 500 (SPX): a stock market index that covers the 500 largest stock exchanges listed in US stock exchanges. The stocks are weighted according to their market capitalization.
- SSE Composite Index (SSEC): a stock market index that covers all stocks that are traded at the Shanghai stock exchange. The stocks are weighted according to their market capitalization.
- DAX Performance Index (DAX): a stock market index that covers 30 major German companies traded at the Frankfurt Stock Exchange.The stocks are weighted according to their market capitalization. The DAX Performance Index also includes dividends.

They can also be seen as approximations of the market portfolios of their respective regions. Even if the experiment might not be arbitrarily scalable to other stocks, fonds or indices, it still covers crucial indicators of the general market movements in the US, Germany and the Shanghai stock exchange.

In figure (6.2) daily annualized volatility between 2000-2020 of the indices SPX, DAX and SSEC is plotted. For the SPX and DAX most notable is the impact of the financial crisis around 2008. For the SSEC the 2015–2016 Chinese stock market turbulence led to the most volatile market. The out of sample forecasting period 01.01.2016 - 01.01.2020 remains relatively calm for all three indices.

Five minute sampled realized volatility requires accurate and complete intraday data to be calculated. While it is more and more common to find this detailed stock data, it can be hard to find for past years. In the Oxford realized library [Link] data for the DAX, SSEC and SPX are available for a time frame from around 2000 until now. Those include the daily realized volatility values used for this experiment. Daily historical data is much more easily available, even though data quality declines with data from 1970 and before. For the three stock indices chosen in this experiment the SPX has the longest available data history. Daily open, close, high and low prices starting from 1970 are easily accessible. For the SSEC and DAX index data is only readily available more recently and was used from the start of 2000. The daily datasets were downloaded from yahoo finance.

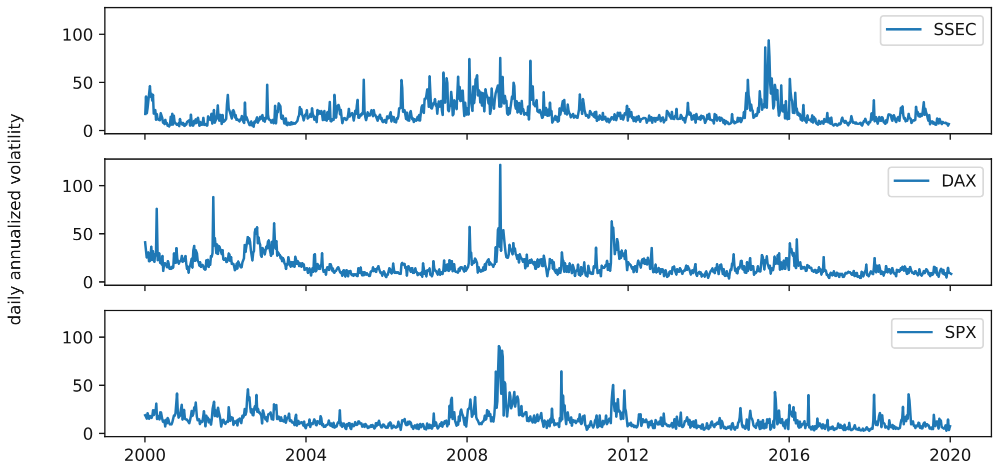

Figure 6.1.: Daily annualized squared returns volatility estimators of the three indices SPX, DAX and SSEC

6.3. LSTM parameters and hyperparameters - training and validation

This section gives detailed explanations how parameters and hyperparameters of the LSTM models were chosen, as well as how the data is split for training.

6.3.1. LSTM - parameters

Firstly we will discuss the choice of input parameters. This is a crucial decision, since it determines how much information we give the model, adjusting the possible ceiling for predictions. The baseline in case of financial time series problems is easily available data like daily high, low, open and closing prices. In Chapter (2) further input variable choices have been discussed. Promising results have been achieved by the inclusion of macroeconomic variables [12] or even sentiment analysis [38]. In this experiment we will stick to using daily high, low, open and closing prices. An interested practitioner might still experiment with adding additional input variables tailored to a specific stock to improve forecasting performance.

Another question arises regarding the question how to scale the input data. If the input variables proportions differ significantly it can lead to slow and faulted learning processes, or even a complete failure in the learning process. This is due to the initial distribution of small random weights. If one input variable is magnitudes larger than another one it is hard to learn which features are important since updating a weight of a small input variable will only yield little changes in the results. This problem can be circumvented by scaling all input data within the same range. Once again there is not one right way to do this.

Common variants are scaling to a range between 0 and 1 or standardization with a mean of 0 and a variance of 1. In this experiment instead of the time series data itself include the series of percentage changes for each input variable.

6.3.2. LSTM - training, validation and test set

In order to properly evaluate a neural networks performance the final model for predictions should not have seen the final test data at any step of the training process. A simple split into a training and test set would yield a prediction, but optimizing hyperparameters on the test set would use out of sample information. In order to avoid this 'peeking' into the future data is generally split into a training, validation and test set. The training set will be used for the in sample calibration of the model, while the validation set is a test set on which hyperparameters can be optimized. Once one has settled on a model with satisfying results for the validation set it can be used for an evaluation of the test set for an unbiased result.

High frequency data is only readily available since the year 2000 at earliest. This restricts training time for the LSTM to a maximum of 20 years. Since common practice is that either a 80/20 or 70/30 split into training and test set should be used to train the model this restricts the time frame for our forecast. To get the maximum amount of training in we choose a four year forecast horizon from 1.1.2016 to 1.1.2020 to test our model, leaving 16 years for training. The 16 years are again split 80/20 into a pure training and a validation set for tuning hyperparameters. GARCH models deliver the best results trained on the longest possible time-frame [13]. In order to simulate a realistic environment where an investor makes use of all data available, GARCH models will always be trained on the maximum available data.

6.3.3. LSTM - hyperparameters

The choice of hyperparameters can have a big influence on a models performance. In a LSTM neural network there are a couple of decisions to be made. For the number of hidden layers, nodes per layer, training epochs and badge size there are no inherently right or wrong choices, usually a mix of rules of thumb and a process of trial and error guides the way to a good prediction.

For the number of hidden layers here we stick to a single one. On the one hand the universal approximation theorem (5.1) ensures that a single hidden layer is able to approximate a wide range of functions. On the other hand empirical results implementing more complex LSTM architectures do not automatically yield improved results, as can be seen in Liu [27] for example.

An epoch refers to the process of going through the whole neural network once. Weights are adjusted after each of those epochs. Through repeating this process weights will be adjusted by a backpropagation logarithm reducing the in sample error. One issue that can arise is that through too many training epochs the data overfits, leading to very precise results for the in-sample prediction but failing to adapt to new out of sample data. To find a balance between overfitting and not recognizing actual patterns we choose an approach with many epochs but also include an early callback function to avoid overfitting. The patience of the early callback function is set to 100, which means that when the validation set error does not improve within the next 100 iterations the training will be stopped and the weights are reset to the epoch with the lowest validation set error.

Optimization of the other two hyperparameters, nodes and batch size, is carried out using the traditional method of performing a grid search. Reasonable sets of integers $\{n_1, n_2, ..., n_{i-1}, n_i\}$ for nodes and $\{b_1, b_2, ..., b_{k-1}, b_k\}$ for batch size, $i, k \in \mathbb{N}$, are chosen. Then every combination of tuples $n_j, b_g,\ j \in \{1, ..., i,\ k \in \{1, ..., k\}$ is tested and the one with the lowest validation set error is chosen.

Every time we train the model from ground up we start with a random set of weights. This leads to different training outcomes even with identical inputs and hyperparameters. In contrast to a deterministic model here we need to validate our results through repetitive execution. After all inputs and hyperparameters are fixed we train the model 10 times and average the test set error. Those averaged errors are proceeded with for comparing them to the statistical benchmark models.

6.4. Implementation

The LSTM model was implemented in Python with Keras. Plots are created with Matplotlib, data preparation was done with Pandas.
The GARCH model was implemented in Python with the ARCH toolbox.
The simple random walk model was also implemented in Python with Pandas.

For code and details on the implementation see Appendix (B).

7. Results

The results of the experiment discussed in chapter (6) are compactly summarized in table (7.1). Errors in the table are obtained by computing the equally weighted MSE averages over each stock index. Tables with separate results for each stock index can be found in Appendix (A).

7.1. single hidden layer LSTM versus benchmark models

Table 7.1.: averaged MSE over all three stock indices

σ_i	Model \ Horizon	2	5	10	20
σ_{r^2}	GARCH	0,635	0,930	1,194	1,888
σ_{r^2}	LSTM	**0,632**	**0,825**	**1,026**	**1,271**
σ_{r^2}	random walk method	0,852	0,985	1,146	1,415
σ_{Park}	GARCH	0,839	1,073	1,379	1,899
σ_{Park}	LSTM	**0,401**	**0,566**	**0,778**	**0,994**
σ_{Park}	random walk method	0,480	0,627	0,810	1,106
σ_{RV5}	GARCH	0,454	0,723	1,045	1,544
σ_{RV5}	LSTM	0,170	**0,245**	**0,336**	**0,470**
σ_{RV5}	random walk method	**0,164**	0,247	0,349	0,518

Overall the LSTM performed quite well against both benchmark models. Notable is the LSTM's superior performance over the random walk method, where in the averaged results of table (7.1) it only once was outperformed slightly in the case of two day forecasts for the realized volatility. Moreover it drastically outperformed the GARCH model for Parkinson and realized volatility estimators, while still slightly trumping the GARCH model for the squared returns estimator as well.

The tables with detailed results for every index can be found in Appendix (A). The results for each individual stock index differ slightly from the averaged results in table (7.1). For the SSE Composite Index the LSTM performed best, only for the two day RV5 forecasts it is narrowly behind the random walk method. Only for the squared returns estimator of the S%P 500 Index the GARCH model was able to show good results and outperformed the LSTM twice for two and ten day horizons. In the case of the DAX Index the LSTM was again only slightly outperformed for the RV5 estimator for two and five day horizons and for the two day squared returns estimator by the GARCH model.

Summarizing the results a pattern can be observed that the LSTM shows consistently better results than the benchmark models for longer horizons. For the short term forecasts the LSTM's results are closer to the benchmark models, and even though they often still yield a lower MSE, sometimes being slightly outperformed by either one of them.

7.2. Comparison of single and multiple hidden layer LSTMs

Table 7.2.: LSTM networks with one, two and three layers for the S&P 500 Index

σ_i	Model \ Horizon	2	5	10	20
σ_{r^2}	LSTM - 1 layer	0,616	0,831	1,069	1,438
σ_{r^2}	LSTM - 2 layer	**0,610**	0,805	**1,028**	**1,410**
σ_{r^2}	LSTM - 3 layer	0,611	**0,798**	1,035	1,413
σ_{Park}	LSTM - 1 layer	0,407	0,618	0,806	1,135
σ_{Park}	LSTM - 2 layer	**0,375**	0,583	**0,779**	**1,131**
σ_{Park}	LSTM - 3 layer	0,385	**0,581**	0,788	1,133
σ_{RV5}	LSTM - 1 layer	**0,169**	**0,266**	**0,401**	**0,572**
σ_{RV5}	LSTM - 2 layer	0,176	0,280	0,411	0,592
σ_{RV5}	LSTM - 3 layer	0,183	0,299	0,431	0,609

In table (7.2) the results for two and three layer LSTM networks are compared to the single layer one. The single layer network has one hidden layer with five nodes, while the second and third hidden layer each have three nodes. At the end of every network there is a dense layer.

The results only differ slightly from the single layer network when adding additional layers. While for the the RV5 volatility estimator the single layer approximation are better for every time horizon the opposite is true for the Parkinson and squared returns estimator. In most cases the difference is within a five percent range. The maximum difference in favor of a multi layer network is the two day prediction for the Parkinson volatility, yielding a 8,5% MSE improvement over the single layer network. On the contrary the single layer network had an 11,1% lower MSE (5 business day prediction versus the) than the multi layer network for the RV5 measure.

Overall it cannot be concluded that more layers generally lower or increase the MSE of forecasts. For the S&P 500 index the results suggest that in case of estimating volatility with the Parkinson or squared returns measure small efficiency gains might be made by adding a second layer, while for the RV5 measure a single layer works best. Since this was only tested for one time span, for one stock index and for one specific architecture choice, those results should not be generalized too light-headedly for other scenarios. Having said this, the overall insight that adding a second layer produces results close to the single layer network and does not show clear improvements is in line with similar experiments as seen in for example Liu [27]. Figures (7.4) and (7.5) depict the behavior of the single and three layer forecast. It is general observed that the multi layer LSTM network will predict closer to the mean and less drastic developments, while the single layer LSTM network tends to follow the recent developments.

Figures (7.1), (7.2), (7.3) and (7.4) depict predictions against the actual values of different S&P 500 volatility estimators and forecasting models. Figure (7.5) depicts the forecast of a three hidden layer LSTM network.

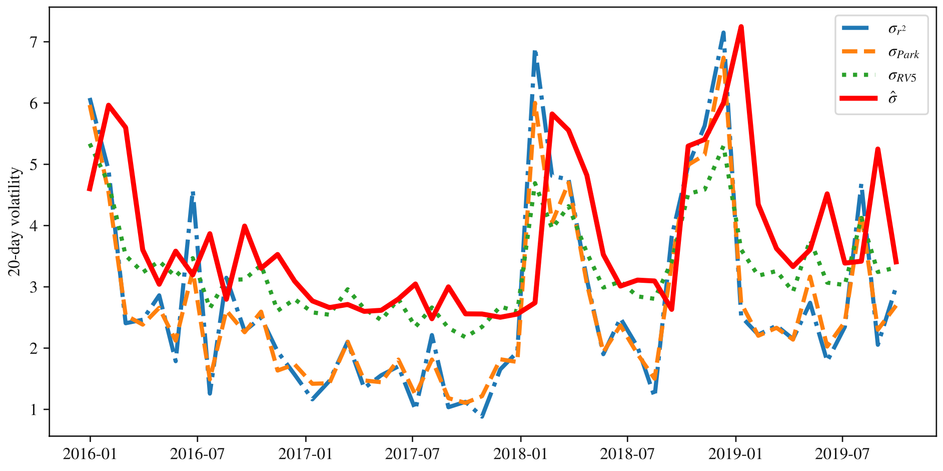

Figure 7.1.: GARCH(1,1) 20 day forecasts compared to all three volatility estimators

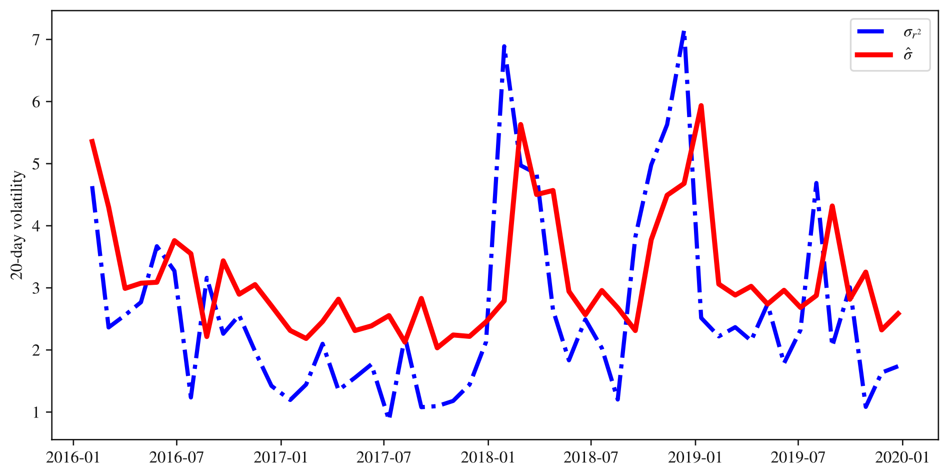

Figure 7.2.: LSTM 20 day forecasts of the squared returns estimator, compared with the actual values

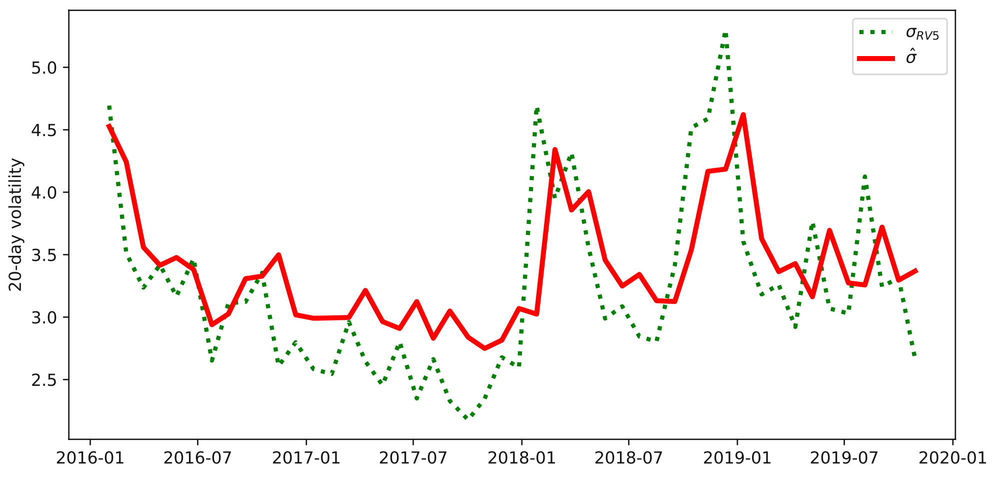

Figure 7.3.: LSTM 20 day forecasts of 5 minute realized volatility, compared with the actual values

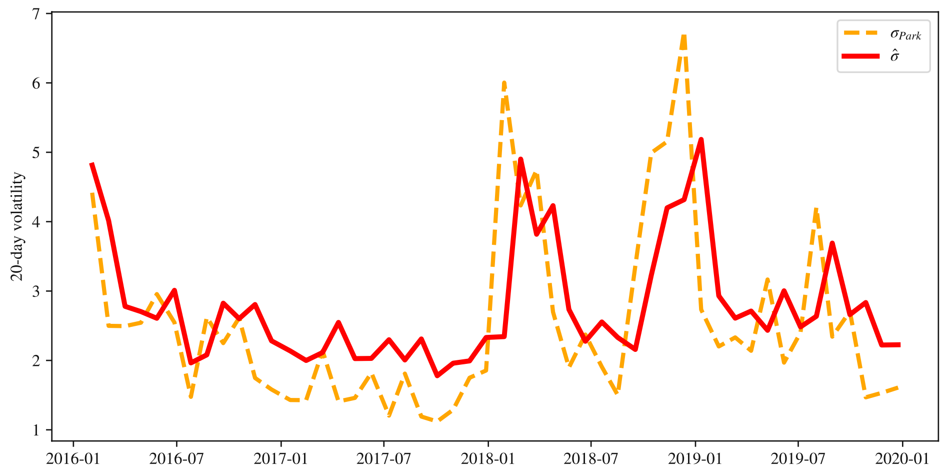

Figure 7.4.: LSTM 20 day forecasts of the Parkinson volatility estimator, compared with the actual values

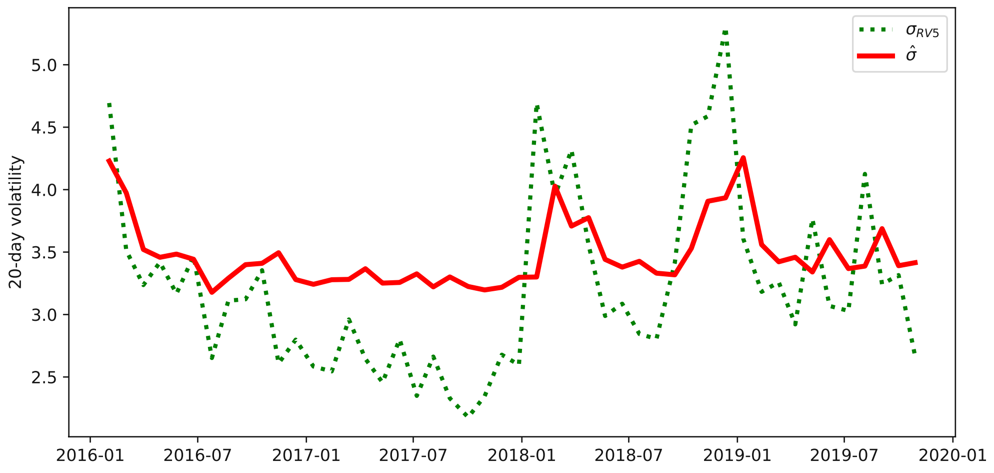

Figure 7.5.: single layer LSTM, 20 day forecasts of 5 minute realized volatility, compared with the actual values of the volatility estimator

8. Conclusion

In chapter (2) it was observed that in some experiments different recurrent neural networks show strong predictive capabilities in volatility forecasting. Another promising result was achieved within the experimental setup of this thesis. Considering different stock indices, time horizons and volatility estimators the LSTM performed well against a GARCH and random walk model. In most cases the LSTM outperformed both benchmark models, only for very short term horizons of two to five business day it sometimes is very slightly outperformed by either the GARCH or random walk model.

The results are in line with many other studies on volatility forecasting, where LSTM networks performed well against econometric models. Due to experimental setup the results should still be interesting, since they show that for a wide range of cases the LSTM is at least on par, and often much better, than the two benchmark models. To the authors knowledge, no comparative experiment similar to this one has been done for a LSTM network yet. Future analysis could include more volatile time periods to examine how a LSTM performs against benchmark models in this environment. Moreover different asset classes such as interest rates, exchange rates or other stocks and bonds could be examined. Another possibility would be to fine tune the LSTM network even more by including exogenous variables, and to test it against state of the art econometric models for each volatility estimate. For a realized measure such as the 5 minute sampled realized volatility the standard GARCH(1,1) model is known to be outperformed by models tailored to the specific situation. In that case a HAR-RV-CJ model for example could prove to be a tougher competitor for a LSTM network.

On a more general note the ceiling for artificial neural network predictions seems quite high. It is easily possible to experiment with a wide range of input variables to tailor a LSTM well to a certain time series problem. In this experiment the decision was made to stick to easily available daily data consisting of high, low, opening and closing prices. As seen in chapter (2) and (5), other exogenous input variables like macroeconomic variables or even related sentiment analysis have already been successfully implemented with improved results over the 'vanilla' LSTM. Also hybrid approaches, including other models predictions as an input for the neural network, show good results. It seems to be a reasonable assumption that some combination of the methods mentioned above and a careful selection of input variables which are already used in classical valuation approaches will improve results even further in the future.

Another direction that could be explored is the LSTM's architecture. Additional layers or gates could be tailored more specifically to fit financial time series data. Moreover it is not set in stone that a LSTM is the ideal structure to tackle volatility predictions. ANNs are still a very active area of research and due to their adaptability and versatility to a wide range of problems it is likely that an even more efficient pattern recognition architecture will yield better results in the future.

A. Additional tables for each stock index separately

Table A.1.: S&P 500 index forecast results

σ_i	Model \ Horizon	2	5	10	20
σ_{r^2}	GARCH	**0,556**	0,844	**0,881**	1,697
σ_{r^2}	LSTM	0,616	**0,831**	1,069	**1,4380**
σ_{r^2}	random walk	0,691	0,875	1,155	1,594
σ_{Park}	GARCH	0,708	0,965	1,296	1,790
σ_{Park}	LSTM	0,407	**0,618**	**0,806**	**1,135**
σ_{Park}	random walk	**0,399**	0,624	0,881	1,307
σ_{RV5}	GARCH	0,320	0,513	0,743	1,082
σ_{RV5}	LSTM	0,170	**0,266**	**0,401**	**0,572**
σ_{RV5}	random walk	**0,169**	0,279	0,414	0,641

Table A.2.: DAX index forecast results

σ_i	Model \ Horizon	2	5	10	20
σ_{r^2}	GARCH	**0,623**	0,891	1,236	1,805
σ_{r^2}	LSTM	0,670	**0,845**	**0,990**	**1,092**
σ_{r^2}	random walk	0,844	0,966	1,095	1,195
σ_{Park}	GARCH	0,852	1,093	1,388	1,892
σ_{Park}	LSTM	**0,401**	**0,512**	**0,711**	**0,770**
σ_{Park}	random walk	0,491	0,581	0,717	0,877
σ_{RV5}	GARCH	0,418	0,659	0,954	1,429
σ_{RV5}	LSTM	0,163	0,220	**0,285**	**0,397**
σ_{RV5}	random walk	**0,153**	**0,209**	0,293	0,418

Table A.3.: SSE Composite Index forecast results

σ_i	Model \ Horizon	2	5	10	20
σ_{r^2}	GARCH	0,727	1,055	1,465	2,163
σ_{r^2}	LSTM	**0,610**	**0,798**	**1,020**	**1,282**
σ_{r^2}	random walk	1,021	1,113	1,188	1,456
σ_{Park}	GARCH	0,958	1,160	1,453	2,014
σ_{Park}	LSTM	**0,396**	**0,569**	**0,816**	**1,078**
σ_{Park}	random walk	0,549	0,677	0,833	1,135
σ_{RV5}	GARCH	0,625	0,997	1,439	2,121
σ_{RV5}	LSTM	0,177	**0,249**	**0,321**	**0,441**
σ_{RV5}	random walk	**0,170**	0,254	0,340	0,494

B. Code

For interested readers or as a starting point for further developments Appendix B contains runnable Python code for the LSTM as well as for the GARCH model. The LSTM was implemented with the Keras package as well as standard tools such as Pandas and Numpy. The GARCH model was implemented with the help of the ARCH package. The Appendix does not contain the complete code for every experiment, it should rather be read as an instruction where different options are explained in comments in between. Nevertheless it is runnable with a suiting set of data. The data here was taken from yahoo finance [https://finance.yahoo.com/, lastly visited 16.02.2021].

In the first code example a three layer LSTM network computes realized volatility estimates for a 20 day horizon. The code is based on different examples on how to implement ANN and LSTM networks in python from Jason Brownlee's website [machinelearningmastery.com, lastly visited 23.03.2021].

```python
import warnings
from math import sqrt
from numpy import concatenate
from matplotlib import pyplot
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from numpy import array
from keras import callbacks
from pandas import DataFrame
from pandas import concat
import matplotlib
import matplotlib.dates as mdates
from datetime import datetime, timedelta
import matplotlib.pyplot as plt
import seaborn as sns
import datetime as dt
import sys
import numpy as np
import pandas as pd
warnings.simplefilter('ignore')

#import all relevant packages

# import relevent data
ox = pd.read_csv(r'C:\Users\david\Diplomarbeit\Daten\oxfordRV_s&p500.csv'
                  , usecols=['Unnamed: 0', 'Symbol', 'rv5', 'open_to_close',
    'open_price', 'close_price'])
ox = ox.rename(columns={'Unnamed: 0': 'Date', 'Symbol': 'Stock', 'rv5': 'rv5
    ', 'open_to_close': 'open_to_close',
                        'open_price': 'open_price', 'close_price': '
    close_price'})
ox['Date'] = ox['Date'].str.slice(0, -15, 1)

# transform into % changes, repeat for every input
perc = np.diff(ox['open_price']) / ox['open_price'][:-1] * 100.
ox['open_price'][0] = 0
ox['open_price'][1::1] = perc

# transform into % changes, repeat for every input
perc = np.diff(ox['close_price']) / ox['close_price'][:-1] * 100.
ox['close_price'][0] = 0
ox['close_price'][1::1] = perc

# use date as index, delete unnecessary columns
spx = ox[ox['Stock'].isin(['.SPX'])]
spx['Date'] = pd.to_datetime(spx['Date'], format='%Y/%m/%d')
spx.set_index('Date', inplace=True)
spx['rv5'] = np.sqrt(spx['rv5']) * 100
spx['20drv5'] = 0
del spx['Stock']

#n business day forecasts will be conducted
n = 20

#calculate the target variable, in this case 20 day five minute sampled
    realized volatility
for i in range((len(spx) - 25)):
    spx['20drv5'].iloc[i] = np.sqrt(np.sum(spx['rv5'][i:i + n]))

#cut to desired length
spx = spx[1:-225]

```

```
#find the right indices
start_val = '2013-1-1'  # y/m/d
end_val = '2016-1-1'  # y/m/d
sd = np.where(spx.index >= start_val)[0].min()
ed = np.where(spx.index >= end_val)[0].min()

#this function reorgnizes the shape of data to suit the keras package
def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
    n_vars = 1 if type(data) is list else data.shape[1]
    df = DataFrame(data)
    cols, names = list(), list()
    for i in range(n_in, 0, -1):
        cols.append(df.shift(i + n - 1))
        names += [('var%d(t-%d)' % (j + 1, i)) for j in range(n_vars)]
    for i in range(0, n_out):
        cols.append(df.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j + 1)) for j in range(n_vars)]
        else:
            names += [('var%d(t+%d)' % (j + 1, i)) for j in range(n_vars)]
    agg = concat(cols, axis=1)
    agg.columns = names
    if dropnan:
        agg.dropna(inplace=True)
    return agg

#convert data with above function
values = spx.values
reframed = series_to_supervised(values, 1, 1)
reframed.drop(reframed.columns[[5, 6, 7, 8]], axis=1, inplace=True)

# split into train and test sets
values = reframed.values
train = values[:sd]
test = values[sd:ed]
# split into input and outputs
train_X, train_y = train[:, :-1], train[:, -1]
test_X, test_y = test[:, :-1], test[:, -1]
# reshape input to be three dimensional [samples, timesteps, features]
train_X = train_X.reshape((train_X.shape[0], 1, train_X.shape[1]))
test_X = test_X.reshape((test_X.shape[0], 1, test_X.shape[1]))
print(train_X.shape, train_y.shape, test_X.shape, test_y.shape)

# network design, here we use three hidden and one dense layer
model = Sequential()
model.add(LSTM(5, return_sequences=True, input_shape=(train_X.shape[1],
    train_X.shape[2])))
model.add(LSTM(3, return_sequences=True))
model.add(LSTM(3))
model.add(Dense(1))
model.compile(loss='mae', optimizer='adam')

# avoid overfitting with early callback if validation loss increases for too
     long
earlystopping = callbacks.EarlyStopping(monitor="val_loss", mode="min",
    patience=50, restore_best_weights=True)

# fit network
history = model.fit(train_X, train_y, epochs=1000, batch_size=4,
    validation_data=(test_X, test_y), verbose=2,
                    shuffle=False, callbacks=[earlystopping])

# summary of the trained model
model.summary()

# make a prediction for the validation set
#the validation set can be used for finetuning hyperparameters
yhat = model.predict(test_X)
test_y = test_y
test_X = test_X.reshape((test_X.shape[0], test_X.shape[2]))

# calculate the MSE for the prediction
rmse = sqrt(mean_squared_error(test_y, yhat))
print('Validation set RMSE: %.3f' % rmse)

# after setting up hyperparameters with the validation set one can make an
    unbiased forecast for the test set
# it is important the test set has not been involved in training! This '
    peeking' would distort results
# make a unbiased test prediction with data not yet seen
train = values[:ed]
train_X = train[:, :-1]
train_y = train[:, -1]
test = values[ed:]
test_X = test[:, :-1]
test_y = test[:, -1]
# reshape input to be 3D [samples, timesteps, features]
train_X = train_X.reshape((train_X.shape[0], 1, train_X.shape[1]))
test_X = test_X.reshape((test_X.shape[0], 1, test_X.shape[1]))

yhat = model.predict(test_X)

```

```
# calculate RMSE
rmse = sqrt(mean_squared_error(test_y, yhat))
print('Test set RMSE: %.3f' % rmse)

# plot the forecast against the actual values for an intuitive comparison
pyplot.plot(spx.index[ed + n::n], test_y[::n], color='green', alpha=0.6,
    linewidth=2.5, linestyle=':',
            label='$\sigma_{RV5}$')
pyplot.plot(spx.index[ed + n::n], yhat[::n], label='$\hat{\sigma}$', color='
    red', linewidth=3, alpha=1)
plt.ylabel("{}-day volatility".format(n))
plt.legend()
pyplot.show()
```

The GARCH model was implemented with the help of the ARCH package in python. Following its documentation a running window forecast was generated. For each day one to n single day variance forecasts are made and used to compute the forecast for the next n-day volatility. See chapters (3) and (4) for details on the computations. Following this the generated forecast is compared to the target variable of a n-day volatility estimator and the MSE is calculated.

```
import warnings
import matplotlib
import matplotlib.dates as mdates
from datetime import datetime, timedelta
warnings.simplefilter('ignore')
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
import sys
import numpy as np
import pandas as pd
from arch import arch_model

#only for plots
import matplotlib
matplotlib.rcParams['mathtext.fontset'] = 'stix'
matplotlib.rcParams['font.family'] = 'STIXGeneral'

#import data
data = pd.read_csv(r'C:\Users\david\Diplomarbeit\Daten\spx_1970-2021.csv')
data['Date'] = pd.to_datetime(data['Date'])
data.set_index('Date', inplace=True)

# prepare data for the ARCH package
market = data['Close']
returns = 100 * market.pct_change().dropna()
```

```
index = returns.index
startdate = '2016-1-1' #y/m/d #start predictions from that day on
n = 20 #n day forecasts
m = 48 #number of n day periods forecasted
k = np.where(index >= startdate)[0].min()
forecasts = {}

#running window forecast, for m*n days
#generates multi day forecasts until n days ahead for each day and saves
    them in an array
for i in range(m*n):
    sys.stdout.write('.')
    sys.stdout.flush()
    am = arch_model(returns, vol='Garch', p=1, o=0, q=1, dist='Normal')
    res = am.fit(first_obs=i, last_obs=i+k, disp='off')
    temp = res.forecast(horizon=n).variance
    fcast = temp.iloc[i + k - 1]
    forecasts[fcast.name] = fcast

fv = pd.DataFrame(forecasts).T
# compute n day volatility for each day
fv['sum'] = np.sqrt(fv.sum(axis=1))

# define other daily volatility measures to compare to the forecast
# Parkinson estimator
data['park'] = (np.sqrt(((np.log(data['High']/(data['Low'])))**2) / (4 * np.
    log(2)))*100)**2
data['park_n_day'] = 0

# volatility estimators computed for next n day period, not including the
    current day
for i in range((len(data)-25)):
    data['park_n_day'].iloc[i] = np.sqrt(np.sum(data['park'][i+1:i+n+1]))

fvi = np.where(data.index >= fv.first_valid_index())[0].min()

#same date
rmse_PARK = np.sqrt(mean_squared_error(fv['sum'], data['park_n_day'][fvi:fvi
    +m*n]))
print('Parkinson volatility: MSE: %.3f' % rmse_PARK)

plt.plot(data['park_n_day'][fvi:fvi+m*n:n], label='$\sigma_{Park}$',alpha=
    0.6,
         linewidth = 2.5, linestyle='--')
plt.plot(fv['sum'][::n], color = 'red', label='$\hat{\sigma}$', linewidth=3,
    alpha= 1)
```

```python
plt.legend()
plt.ylabel("{}-day volatility".format(n))
plt.show()
```

Bibliography

[1] Torben Andersen et al. *Modeling and Forecasting Realized Volatility.* en. Tech. rep. w8160. Cambridge, MA: National Bureau of Economic Research, Mar. 2001, w8160. DOI: 10.3386/w8160. URL: http://www.nber.org/papers/w8160.pdf (visited on 04/09/2021).

[2] Torben G. Andersen et al. „Chapter 15 Volatility and Correlation Forecasting". en. In: *Handbook of Economic Forecasting.* Vol. 1. Elsevier, 2006, pp. 777–878. ISBN: 978-0-444-51395-3. DOI: 10.1016/S1574-0706(05)01015-3. URL: https://linkinghub.elsevier.com/retrieve/pii/S1574070605010153 (visited on 04/09/2021).

[3] Josip Arnerić, Tea Poklepović, and Zdravka Aljinović. „GARCH based artificial neural networks in forecasting conditional variance of stock returns". en. In: *Croatian Operational Research Review* 5.2 (Dec. 2014), pp. 329–343. ISSN: 18480225, 18489931. DOI: 10.17535/crorr.2014.0017. URL: http://hrcak.srce.hr/index.php?show=clanak&id_clanak_jezik=197475&lang=en (visited on 04/09/2021).

[4] Ole E. Barndorff-Nielsen et al. „Subsampling realised kernels". In: *Journal of Econometrics* 160.1 (2011). Realized Volatility, pp. 204–219. ISSN: 0304-4076. DOI: https://doi.org/10.1016/j.jeconom.2010.03.031. URL: https://www.sciencedirect.com/science/article/pii/S0304407610000734.

[5] Bernard Bollen and Brett Inder. „Estimating daily volatility in financial markets utilizing intraday data". en. In: *Journal of Empirical Finance* 9.5 (Dec. 2002), pp. 551–562. ISSN: 09275398. DOI: 10.1016/S0927-5398(02)00010-5. URL: https://linkinghub.elsevier.com/retrieve/pii/S0927539802000105 (visited on 04/09/2021).

[6] Tim Bollerslev. „Generalized autoregressive conditional heteroskedasticity". en. In: *Journal of Econometrics* 31.3 (Apr. 1986), pp. 307–327. ISSN: 03044076. DOI: 10.1016/0304-4076(86)90063-1. URL: https://linkinghub.elsevier.com/retrieve/pii/0304407686900631 (visited on 04/09/2021).

[7] Andrea Bucci. „Realized Volatility Forecasting with Neural Networks". en. In: *Journal of Financial Econometrics* 18.3 (June 2020), pp. 502–531. ISSN: 1479-8409, 1479-8417. DOI: 10.1093/jjfinec/nbaa008. URL: https://academic.oup.com/jfec/article/18/3/502/5856840 (visited on 04/09/2021).

[8] F. Corsi. „A Simple Approximate Long-Memory Model of Realized Volatility". en. In: *Journal of Financial Econometrics* 7.2 (Nov. 2008), pp. 174–196. ISSN: 1479-8409, 1479-8417. DOI: 10.1093/jjfinec/nbp001. URL: https://academic.oup.com/jfec/article-lookup/doi/10.1093/jjfinec/nbp001 (visited on 05/15/2021).

[9] G. Cybenko. „Approximation by superpositions of a sigmoidal function“. en. In: *Mathematics of Control, Signals, and Systems* 2.4 (Dec. 1989), pp. 303–314. ISSN: 0932-4194, 1435-568X. DOI: 10.1007/BF02551274. URL: http://link.springer.com/10.1007/BF02551274 (visited on 06/18/2021).

[10] Robert F. Engle. „Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation“. In: *Econometrica* 50.4 (July 1982), p. 987. ISSN: 00129682. DOI: 10.2307/1912773. URL: https://www.jstor.org/stable/1912773?origin=crossref (visited on 04/09/2021).

[11] William Feller. „The Asymptotic Distribution of the Range of Sums of Independent Random Variables“. en. In: *The Annals of Mathematical Statistics* 22.3 (Sept. 1951), pp. 427–432. ISSN: 0003-4851. DOI: 10.1214/aoms/1177729589. URL: http://projecteuclid.org/euclid.aoms/1177729589 (visited on 06/14/2021).

[12] Marcelo Fernandes, Marcelo C. Medeiros, and Marcel Scharth. „Modeling and predicting the CBOE market volatility index“. en. In: *Journal of Banking & Finance* 40 (Mar. 2014), pp. 1–10. ISSN: 03784266. DOI: 10.1016/j.jbankfin.2013.11.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0378426613004172 (visited on 04/09/2021).

[13] Stephen Figlewski. „Forecasting Volatility“. In: *Financial Markets, Institutions & Instruments* 6.1 (1997), pp. 1–88. DOI: https://doi.org/10.1111/1468-0416.00009. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1468-0416.00009. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1468-0416.00009.

[14] Christos Floros et al. „Realized Measures to Explain Volatility Changes over Time“. en. In: *Journal of Risk and Financial Management* 13.6 (June 2020), p. 125. ISSN: 1911-8074. DOI: 10.3390/jrfm13060125. URL: https://www.mdpi.com/1911-8074/13/6/125 (visited on 04/09/2021).

[15] Mark B. Garman and Michael J. Klass. „On the Estimation of Security Price Volatilities from Historical Data“. en. In: *The Journal of Business* 53.1 (Jan. 1980), p. 67. ISSN: 0021-9398, 1537-5374. DOI: 10.1086/296072. URL: https://www.jstor.org/stable/2352358 (visited on 04/09/2021).

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2016. ISBN: 978-0-262-03561-3.

[17] Nicole Gruber and Alfred Jockisch. „Are GRU Cells More Specific and LSTM Cells More Sensitive in Motive Classification of Text?“ In: *Frontiers in Artificial Intelligence* 3 (June 2020), p. 40. ISSN: 2624-8212. DOI: 10.3389/frai.2020.00040. URL: https://www.frontiersin.org/article/10.3389/frai.2020.00040/full (visited on 06/18/2021).

[18] Trino-Manuel Ñíguez. „Evaluating monthly volatility forecasts using proxies at different frequencies“. In: *Finance Research Letters* 17 (2016), pp. 41–47. ISSN: 1544-6123. DOI: https://doi.org/10.1016/j.frl.2016.01.008. URL: https://www.sciencedirect.com/science/article/pii/S154461231600009X.

[19] Peter R. Hansen and Asger Lunde. „A forecast comparison of volatility models: does anything beat a GARCH(1,1)?" en. In: *Journal of Applied Econometrics* 20.7 (Dec. 2005), pp. 873–889. ISSN: 0883-7252, 1099-1255. DOI: 10.1002/jae.800. URL: http://doi.wiley.com/10.1002/jae.800 (visited on 04/09/2021).

[20] Steven L. Heston. „A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options". en. In: *Review of Financial Studies* 6.2 (Apr. 1993), pp. 327–343. ISSN: 0893-9454, 1465-7368. DOI: 10.1093/rfs/6.2.327. URL: https://academic.oup.com/rfs/article-lookup/doi/10.1093/rfs/6.2.327 (visited on 04/10/2021).

[21] Sepp Hochreiter and Jürgen Schmidhuber. „Long Short-Term Memory". en. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667, 1530-888X. DOI: 10.1162/neco.1997.9.8.1735. URL: https://direct.mit.edu/neco/article/9/8/1735-1780/6109 (visited on 04/09/2021).

[22] Kurt Hornik. „Approximation capabilities of multilayer feedforward networks". en. In: *Neural Networks* 4.2 (1991), pp. 251–257. ISSN: 08936080. DOI: 10.1016/0893-6080(91)90009-T. URL: https://linkinghub.elsevier.com/retrieve/pii/089360809190009T (visited on 06/18/2021).

[23] Yan Hu, Jian Ni, and Liu Wen. „A hybrid deep learning approach by integrating LSTM-ANN networks with GARCH model for copper price volatility prediction". en. In: *Physica A: Statistical Mechanics and its Applications* 557 (Nov. 2020), p. 124907. ISSN: 03784371. DOI: 10.1016/j.physa.2020.124907. URL: https://linkinghub.elsevier.com/retrieve/pii/S0378437120304696 (visited on 04/09/2021).

[24] Paraskevi Katsiampa. „Volatility estimation for Bitcoin: A comparison of GARCH models". en. In: *Economics Letters* 158 (Sept. 2017), pp. 3–6. ISSN: 01651765. DOI: 10.1016/j.econlet.2017.06.023. URL: https://linkinghub.elsevier.com/retrieve/pii/S0165176517302501 (visited on 04/09/2021).

[25] Werner Kristjanpoller R and Esteban Hernández P. „Volatility of main metals forecasted by a hybrid ANN-GARCH model with regressors". en. In: *Expert Systems with Applications* 84 (Oct. 2017), pp. 290–300. ISSN: 09574174. DOI: 10.1016/j.eswa.2017.05.024. URL: https://linkinghub.elsevier.com/retrieve/pii/S0957417417303408 (visited on 04/09/2021).

[26] Lily Y. Liu, Andrew J. Patton, and Kevin Sheppard. „Does Anything Beat 5-Minute RV? A Comparison of Realized Measures Across Multiple Asset Classes". In: *Journal of Econometrics* 187.1 (2015), pp. 293–311. ISSN: 0304-4076. DOI: https://doi.org/10.1016/j.jeconom.2015.02.008. URL: https://www.sciencedirect.com/science/article/pii/S0304407615000329.

[27] Yang Liu. „Novel volatility forecasting using deep learning–Long Short Term Memory Recurrent Neural Networks". en. In: *Expert Systems with Applications* 132 (Oct. 2019), pp. 99–109. ISSN: 09574174. DOI: 10.1016/j.eswa.2019.04.038. URL: https://linkinghub.elsevier.com/retrieve/pii/S0957417419302635 (visited on 04/09/2021).

[28] Yifan Liu et al. „Stock Volatility Prediction Using Recurrent Neural Networks with Sentiment Analysis“. In: *arXiv:1705.02447 [cs]* (May 2017). arXiv: 1705.02447. URL: http://arxiv.org/abs/1705.02447 (visited on 04/11/2021).

[29] Jose A. Lopez. „Evaluating the predictive accuracy of volatility models“. In: *Journal of Forecasting* 20.2 (2001), pp. 87–109. DOI: https://doi.org/10.1002/1099-131X(200103)20:2<87::AID-FOR782>3.0.CO;2-7. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/1099-131X%28200103%2920%3A2%3C87%3A%3AAID-FOR782%3E3.0.CO%3B2-7. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/1099-131X%28200103%2920%3A2%3C87%3A%3AAID-FOR782%3E3.0.CO%3B2-7.

[30] Leandro Maciel, Fernando Gomide, and Rosangela Ballini. „Evolving Fuzzy-GARCH Approach for Financial Volatility Modeling and Forecasting“. en. In: *Computational Economics* 48.3 (Oct. 2016), pp. 379–398. ISSN: 0927-7099, 1572-9974. DOI: 10.1007/s10614-015-9535-2. URL: http://link.springer.com/10.1007/s10614-015-9535-2 (visited on 04/09/2021).

[31] Daniel B. Nelson. „Conditional Heteroskedasticity in Asset Returns: A New Approach“. In: *Econometrica* 59.2 (Mar. 1991), p. 347. ISSN: 00129682. DOI: 10.2307/2938260. URL: https://www.jstor.org/stable/2938260?origin=crossref (visited on 04/09/2021).

[32] Michael Parkinson. „The Extreme Value Method for Estimating the Variance of the Rate of Return“. en. In: *The Journal of Business* 53.1 (Jan. 1980), p. 61. ISSN: 0021-9398, 1537-5374. DOI: 10.1086/296071. URL: https://www.jstor.org/stable/2352357 (visited on 04/09/2021).

[33] Razvan Pascanu, Tomás Mikolov, and Yoshua Bengio. „Understanding the exploding gradient problem“. In: *CoRR* abs/1211.5063 (2012). arXiv: 1211.5063. URL: http://arxiv.org/abs/1211.5063.

[34] Gábor Petneházi and József Gáll. „Exploring the predictability of range-based volatility estimators using RNNs“. en. In: *arXiv:1803.07152 [q-fin, stat]* (Mar. 2018). arXiv: 1803.07152. URL: http://arxiv.org/abs/1803.07152 (visited on 04/09/2021).

[35] Ser-Huang Poon. „A Practical Guide to Forecasting Financial Market Volatility“. en. In: *Financial Market Volatility* (Jan. 2005), p. 238.

[36] L. C. G. Rogers and S. E. Satchell. „Estimating Variance From High, Low and Closing Prices“. In: *The Annals of Applied Probability* 1.4 (Nov. 1991). ISSN: 1050-5164. DOI: 10.1214/aoap/1177005835. URL: https://projecteuclid.org/journals/annals-of-applied-probability/volume-1/issue-4/Estimating-Variance-From-High-Low-and-Closing-Prices/10.1214/aoap/1177005835.full (visited on 04/09/2021).

[37] L. C. G. Rogers, S. E. Satchell, and Y. Yoon. „Estimating the volatility of stock prices: a comparison of methods that use high and low prices“. en. In: *Applied Financial Economics* 4.3 (June 1994), pp. 241–247. ISSN: 0960-3107, 1466-4305. DOI: 10.1080/758526905. URL: http://www.tandfonline.com/doi/abs/10.1080/758526905 (visited on 04/09/2021).

[38] Marcelo Sardelich and Suresh Manandhar. „Multimodal deep learning for short-term stock volatility prediction“. In: *arXiv:1812.10479 [cs, q-fin, stat]* (Dec. 2018). arXiv: 1812.10479. URL: http://arxiv.org/abs/1812.10479 (visited on 04/09/2021).

[39] Jacopo De Stefani et al. „Machine Learning for Multi-step Ahead Forecasting of Volatility Proxies“. en. In: MIDAS@ PKDD/ECML (), pp. 17–28.

[40] Dennis Yang and Qiang Zhang. „Drift Independent Volatility Estimation Based on High, Low, Open, and Close Prices“. en. In: *The Journal of Business* 73.3 (July 2000), pp. 477–492. ISSN: 0021-9398, 1537-5374. DOI: 10.1086/209650. URL: https://www.jstor.org/stable/10.1086/209650 (visited on 04/09/2021).

Eidesstattliche Erklärung

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

Wien, am 6.8.2021 [illegible]