THM TECHNISCHE HOCHSCHULE MITTELHESSEN
CAMPUS FRIEDBERG
MND Mathematik, Naturwissenschaften und Datenverarbeitung

Bestimmung der Güte verschiedener Prognose-Modelle für Aktienkursverläufe

Projektbericht

Studiengang	Business Mathematics
Modul	Projekt Simulation
Professor	Prof. Dr. Sven Oliver Hein
Referent	Antonio Beslic, Büsra Karaoglan

Friedberg, im August 2018

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Im Rahmen des Moduls MA6011 *Projekt Simulation* werden wir untersuchen, inwiefern sich Aktienkursverläufe mit Daten aus der Vergangenheit prognostizieren lassen. Auf den ersten Blick scheint eine Vorhersage unmöglich zu sein, da die Verläufe von Aktienkursen einen unberechenbaren Eindruck machen. Zudem gibt es unzählige Einflüsse in unserer komplexen Welt, welche den zukünftigen Werteverlauf eines Aktienkurses beeinflussen können.

Um desto trotz eine Vorhersage machen zu können, müssen wir uns bei der Modellierung auf die wesentlichen Einflussfaktoren, die den Verlauf eines Aktienkurses bestimmen, einschränken. Dabei sollen vereinfachte Modelle eingeführt werden, welche für die Prognose lediglich vergangene Zeitreihen verwenden. Aus diesen Zeitreihen lassen sich beispielsweise *Volatilität* oder auch *Mittelwert* eines Aktienkurses bestimmen, welche Grundlage für die darauffolgenden Simulationen sind. Um über die Qualität des Modells zu urteilen, werden bezüglich einer realen Zeitreihe mehrere Simulationen durchgeführt. Anschließend sollen uns Gütemaße helfen, die verschiedenen Modelle zu vergleichen und gegenüberzustellen. Beispielhaft erklärt: Wir schauen uns die Facebook-Schlusskurse der letzten zwei Jahre an. Anhand der Daten aus dem ersten Jahr, soll der Verlauf im zweiten Jahr simuliert werden. Im Nachhinein wird der simulierte und tatsächliche Verlauf mithilfe von Gütemaßen verglichen.

Im zweiten Kapitel werden wir die zu untersuchenden Modelle vorstellen. Dabei wollen wir über die Sinnhaftigkeit der Modelle diskutieren und den dazugehörigen *Mathematica*©[1] bzw. *Python*[2] Code näher erläutern. Anschließend werden wir im dritten Kapitel verschiedene Gütemaße zur Bewertung unserer Modellprognosen vorstellen, bevor wir im vierten Kapitel unsere Modelle bewerten und unsere Ergebnisse vorstellen. Im letzten Kapitel werden wir unsere Arbeit mit einem zusammenfassenden Fazit und einen Ausblick auf weitere Untersuchungen beenden.

2 Prognose-Modelle

In diesem Kapitel werden verschiedene Prognose Modelle einleitend vorgestellt. Diese wollen wir mithilfe der Kurse vom IBM testen. Die Daten haben wir dankenswerter Weise von der Walter Ludwig Wertpapierhandelsbank zur Verfügung gestellt bekommen. Die IBM Kurse haben wir in Form von Tages-Schlusskursen, Stunden-, Minuten-, sowie Sekundenkurse gegeben. Im Rahmen dieser Arbeit werden wir ausschließlich die Tagesschlusskurse zur Prognose verwenden. Dabei konzentrieren wir uns auf Langzeitprognosen, da weitere Betrachtungen den Rahmen dieser Arbeit sprengen würden. Im Abschnitt 2.5.3 werden wir jedoch beispielhaft eine Kurzzeitprognose mit neuronalen Netzen durchführen.

Das Einlesen der Daten wird im Anhang A erläutert. Im Folgenden sind die Datensätze wie folgt benannt und definiert: KursTage17 (Daten aus dem gesamten Jahr 2017) und KursTage18 (Daten vom Januar 2018 bis April 2018).

2.1 M1: Random Walk ohne Drift

Das erste Modell, welches wir vorstellen wollen ist ein Random Walk ohne Drift. Wie der Name des Modells vermuten lässt, geht es bei Random Walks um die Modellierung eines Prozesses, bei den jeder einzelne Schritt zufällig erfolgt.

Definition 2.1 (Eindimensionaler Random Walk ohne Drift). Gegeben sei eine Folge von unabhängigen Zufallsvariablen $Z_1, Z_2, ...$ mit Werten in $\mathbb{Z}$. Besitzen die Zufallsvariablen alle die gleiche

[1] https://www.wolfram.com/mathematica/
[2] https://www.python.org/

Verteilung, so bezeichne man den folgenden definierten stochastischen Prozess

$$X_n = X_0 + \sum_{j=1}^{n} Z_j, \quad n \in \mathbb{N}_0$$

als eindimensionalen Random Walk.

Schauen wir uns ein Beispiel dazu an. Die womöglich einfachste Veranschaulichung eines Random Walks ist der Münzwurf. Dabei soll in diesem Fall $X_0 = 0$ der Ausgangspunkt sein. Die Zufallsvariablen sollen den Wert 1 annehmen, falls Kopf fällt und -1, falls Zahl fällt. Die Münze ist zudem fair, sodass die Wahrscheinlichkeit für beide Variablen zu jedem Zeitpunkt 50% beträgt. Je nachdem, ob Kopf oder Zahl fällt, bewegen wir uns demnach im Koordinatensystem nach oben rechts oder nach unten rechts. In Mathematica lässt sich eine derartige Irrfahrt recht einfach darstellen. Dafür geben wir uns mit der Mathematica-Funktion `RandomInteger[]` eine Liste n Bernoulli-verteilter Zufallsvariablen mit den Werten 0 und 1 aus. Der Wert 0 stellt in diesem Fall den Kopf der Münze und 1 die Zahl der Münze dar. Die Simulation sieht umgesetzt so aus:

```
EinfacheIrrfahrtGraphik[Startwert_, n_] :=
Module[{y = Startwert, list1 = RandomInteger[{0, 1}, n], list3 = {}},
For[i = 1, i <= n, i++, If[list1[[i]] == 0, y = y - 1, y = y + 1];
AppendTo[list3, y]]; ListLinePlot[list3]]

EinfacheIrrfahrtGraphik[0, 10000]
Out:=
```

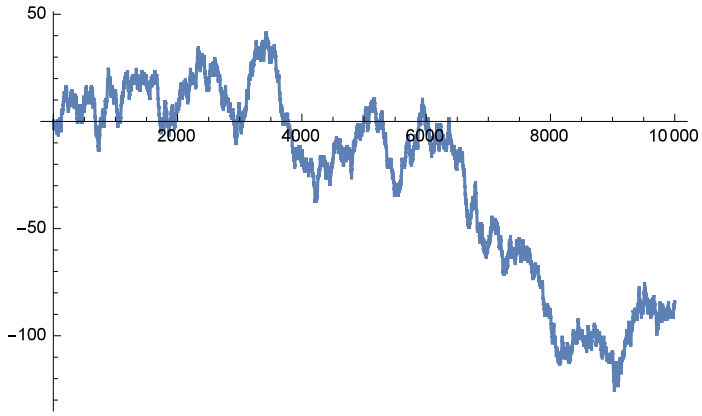

Abbildung 1: Beispiel eines Random Walks

Dass in dieser Simulation X_{10000} einen Wert um -80 hat, ist zufällig.

Bisher haben wir nur additive Irrfahrten betrachtet. Ein Beispiel für eine multiplikative Irrfahrt ergibt sich z.B. durch folgende Rekursionformel

$$y_t = y_0 \cdot \prod_{k=1}^{t} (1 + r_k), \quad \text{für } t \in [0; t_{max}]$$

wobei r_t unabhängig identisch verteilte Renditen darstellen mit $r_t = 0.005 \cdot \eta_t, \quad (\eta_t \sim \mathcal{N}(0,1))$ Betrachte man den Faktor 0.005 als eine Art Volatilität der Aktie, so lässt sich der Aktienkursverlauf bzw. die multiplikative Irrfahrt folglich darstellen:

```
Irrfahrt1[ini_, tmax_] :=
Module[{listvalues = {}},
multipliers = 1 + RandomVariate[NormalDistribution[], tmax]*0.001;
For[i = 1, i <= tmax, i++,
values = ini*Product[multipliers[[j]], {j, 1, i}];
listvalues = AppendTo[listvalues, values]];
```

```
Return[ListLinePlot[listvalues]]]

Irrfahrt1[100, 100]
Out:=
```

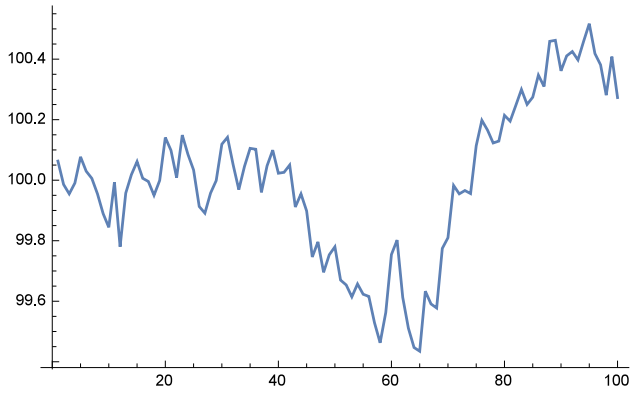

Abbildung 2: Beispiel einer multiplikativen Irrfahrt

Da sich jedoch mit log-Renditen bekanntlich einfacher rechnen lässt, kann eine Rekursionsformel auch so aussehen:

$$y_t = y_0 \cdot \exp\left(\sum_{k=1}^{t} r_k\right), \quad \text{für } t \in [0; t_{max}] \tag{1}$$

Die Umsetzung in Mathematica kann wie folgt aussehen:

```
Irrfahrt2[ini_, tmax_, volatility_] :=
Module[{listvalues = {}, vol = volatility},
r = RandomVariate[NormalDistribution[], tmax + 1]*vol;
For[i = 1, i <= tmax, i++, values = ini*Exp[Sum[r[[j]], {j, 1, i}]];
listvalues = AppendTo[listvalues, values]];
Return[ListLinePlot[listvalues,
PlotStyle -> {RandomColor[], Thick}]]]

Irrfahrt2[100, 100, 0.01]
Out:=
```

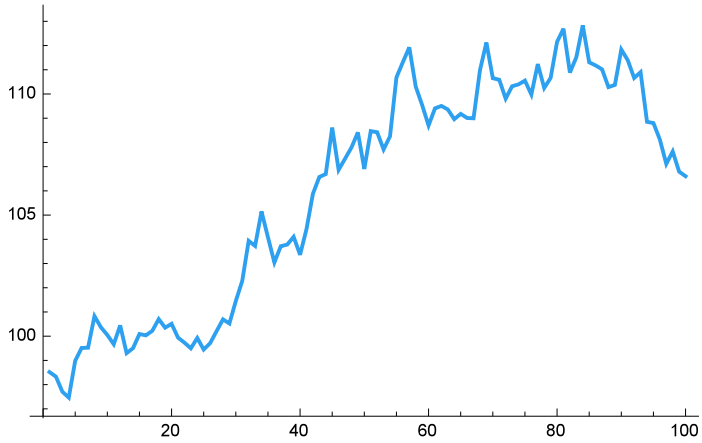

Abbildung 3: Beispiel einer multiplikativen Irrfahrt mit log-Renditen

Der *Mathematica* Code, welcher auf Grundlage dieses Modells und den IBM Daten aus dem Jahr 2017 eine Prognose simulieren soll, sieht folgendermaßen aus:

```
Modell1[ini_, tmax_, volatility_, muT_] :=
Module[{vol = volatility, mu = muT},
r = RandomVariate[NormalDistribution[mu, vol], tmax + 1];
Table[ini*Exp[Sum[r[[j]], {j, 1, i}]], {i, 1, tmax}]]
```

Im Modul geben wir den Startwert, die Anzahl an Prognoseschritten, die Volatilität und den Mittelwert an.

Da wir auf Grundlage der Daten aus 2017 die Daten aus 2018 schätzen wollen, ist unser Startwert `ini` immer der Kurs des letzten Handelstages aus dem Jahr 2017. Die Anzahl der Prognoseschritte `tmax` soll dann die Anzahl der Handelstage zwischen Januar 2018 bis April 2018 sein, sodass wir die prognostizierten Kurse mit den realisierten Kursen vergleichen können. Als Volatilität `volatility` und Mittelwert `muT` nehmen wir die Standardabweichung und den Mittelwert der logarithmierten Renditen aus dem Jahr 2017.

Nach der erfolgreichen Eingabe erhält man einen simulierten Pfad für die Tagesschlusskurse, welchen man sich plotten kann.

```
logTage17 = Table[Log[KursTage17[[j]]/KursTage17[[j - 1]]], {j, 2, Length[KursTage17]}];

muTage17 = Mean[logTage17];

sigmaTage17 = StandardDeviation[logTage17];

plotModell1 =
ListLinePlot[Modell1[Last[KursTage17], Length[KursTage18], sigmaTage17, muTage17],
PlotStyle -> RandomColor[]]

Out:=
```

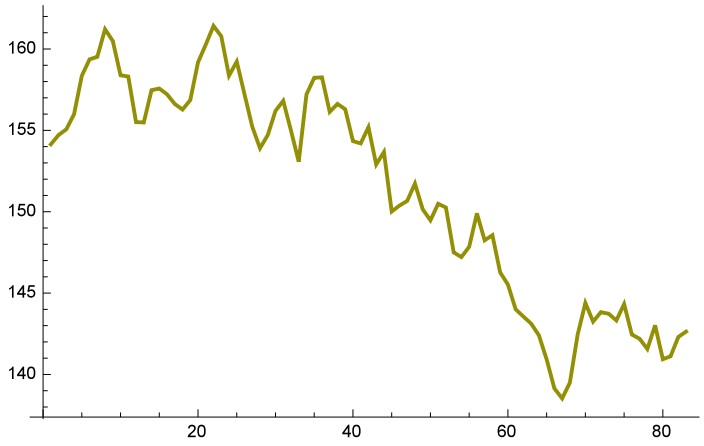

Abbildung 4: Simulierter Pfad mit Modul `Modell1`

Nachdem die logarithmierten Renditen samt Standardabweichung und Mittelwert aus dem Jahr 2017 berechnet wurden, trägt man diese im Modul ein. Den letzten Handelstag aus dem Jahr 2017 erhält man mit `Last[KursTage17]` und den Eingabewert `tmax` initialisieren wir mit der Anzahl der Handelstage von Januar 2018 bis April 2018, also `Length[KursTage18]`.

Wir nehmen hierbei an, dass die logarithmierten Renditen normalverteilt sind. So erzeugen wir insgesamt `tmax` normalverteilte Zufallszahlen mit den Parametern $\mu =$ `muTage17` und $\sigma =$ `sigmaTage17`. Danach werden die simulierten Kurse wie bei der multiplikativen Irrfahrt in Formel (1) berechnet. Als Ausgabe erhält man den simulierten Pfad als Plot.

Vergleichen wir den simulierten Pfad aus Abbildung 4 mit den tatsächlichen Verlauf der Schlusskurse im Jahr 2018.

```
Show[plotModell1, ListLinePlot[KursTage18], PlotRange -> All]

Out:=
```

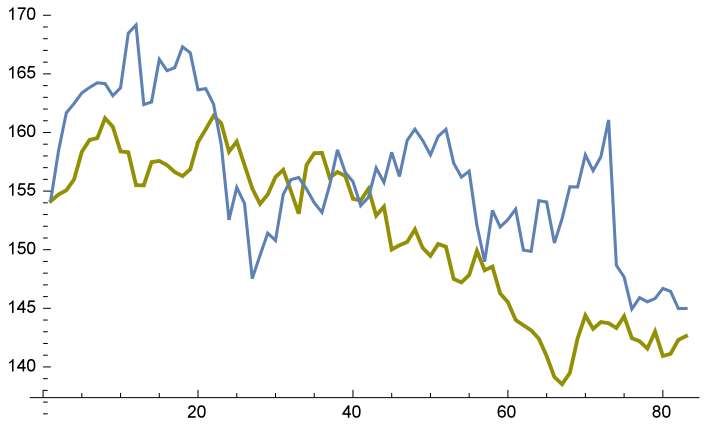

Abbildung 5: Vergleich des simulierten Pfades (gelb) des Moduls `Modell1` mit tatsächlichem Verlauf (blau)

Anhand eines simulierten Pfades kann man natürlich keinerlei Aussagen über die Güte des Modells treffen. So werden wir im Ergebnisteil eine große Anzahl an Pfade simulieren und den daraus resultierenden Mittelwert der Pfade als Prognosepfad interpretieren (Monte-Carlo-Simulation (vgl. [2, Seite 2])).

Nebenbei wollen wir anmerken, dass die Annahme der normalverteilten log-Renditen sehr vage ist. Vergleicht man folglich die Verteilung der wahren Renditen mit den simulierten Renditen:

```
Show[SmoothHistogram[logTage17, PlotRange -> All,
PlotStyle -> Orange],
Plot[PDF[NormalDistribution[muTage17, sigmaTage17], x], {x, -0.04,
0.04}]]

Out:=
```

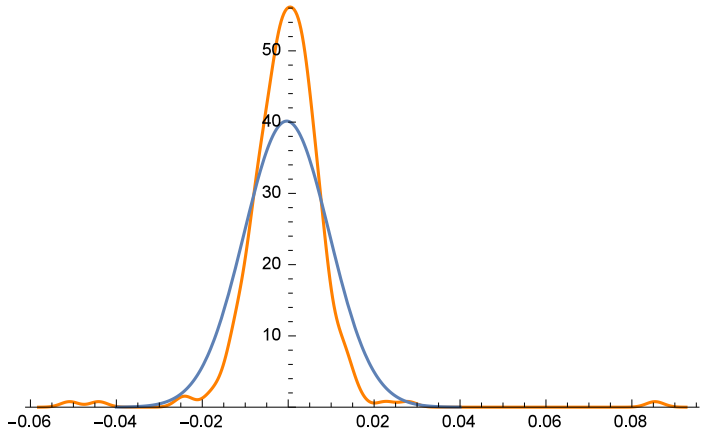

Abbildung 6: Vergleich der Verteilung der log-Renditen (orange) mit Normalverteilung (blau)

Man erkennt deutlich, dass die angenommene Normalverteilung alles andere als zufriedenstellend ist. Der *Mathematica* Befehl `DistributionFitTest` überprüft formal mittels mehrerer Test-Statistiken, ob die log-Renditen normalverteilt sind:

```
DistributionFitTest[logTage17, NormalDistribution[muTage17, sigmaTage17], "ShortTestConclusion"]
```

Als Ausgabewert erhält man den Wert `Reject`, sodass die Annahme verworfen werden kann. Insgesamt werden intern mit diesem Befehl zwölf Test-Statistiken ausgewertet. Alle dieser Tests verwerfen diese Annahme, welche man in folgender Tabelle erkennen kann.

```
DistributionFitTest[logTage17, Automatic,
"HypothesisTestData"]["PValueTable", All]
```

Test-Statistik	p-Wert
Anderson-Darling	2.6125×10^{-8}
Baringhaus-Henze	7.8507×10^{-7}
Cramér-von Mises	$0.$
Jarque-Bera ALM	$0.$
Kolmogorov-Smirnov	$0.$
Kuiper	8.8952×10^{-9}
Mardia Combined	$0.$
Mardia Kurtosis	1.0555×10^{-1382}
Mardia Skewness	1.1955×10^{-28}
Pearson $\mathcal{X}^2$	5.1753×10^{-44}
Shapiro-Wilk	9.5561×10^{-18}
Watson U	$0.$

Tabelle 1: Test-Statistiken zur Überprüfung der Annahme von normalverteilten log-Renditen

Der p-Wert ist in jedem Test 0 bzw. so gut wie 0. Im folgenden Abschnitt 2.2 werden wir eine bessere Verteilungsannahme treffen. Die Prognose müsste damit besser sein.

2.2 M2: Random Walk mit angepasster Verteilungsannahme

Durch die Test-Statistiken in Tabelle 1 konnten wir uns deutlich machen, dass die Annahme der normalverteilten log-Renditen falsch ist. Um das vorige Modell zu verbessern, werden wir eine Verteilung finden, welche unsere log-Renditen besser annähert. Mithilfe des *Mathematica* Befehls `FindDistribution[]` erhalten wir einen Vorschlag. Für unsere Tages-log-Renditen erhalten wir die studentsche t-Verteilung als Vorschlag. Nun können wir die drei Parameter der studentschen t-Verteilung μ ,σ und ν an unsere Daten fitten.

```
H = DistributionFitTest[logTage17, StudentTDistribution[mu1, sig1, nu1], "HypothesisTestData"];
H["FittedDistributionParameters"]

Out:= {mu1 -> -0.000243831, sig1 -> 0.0057266, nu1 -> 3.41837}
```

Vergleichen wir jetzt analog zu Abbildung 6 die Verteilung der tatsächlichen log-Renditen mit der studentschen t-Verteilung.

```
Fit1 = H["FittedDistribution"];
Show[Plot[PDF[Fit1, x], {x, -0.04, 0.04}, PlotRange -> All],
SmoothHistogram[logTage17, PlotStyle -> Orange, PlotRange -> All]]

Out:=
```

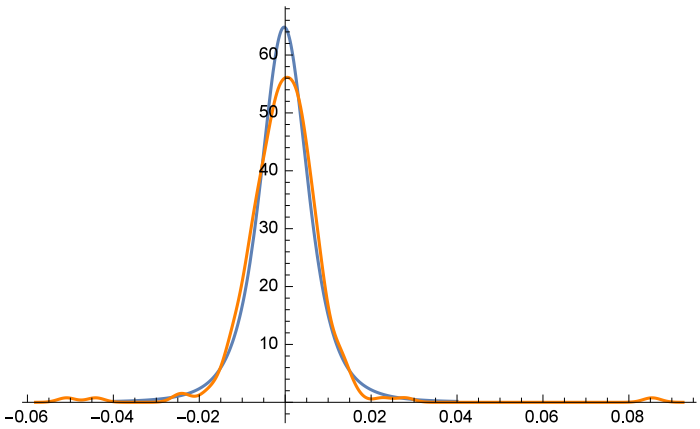

Abbildung 7: Vergleich der Verteilung der log-Renditen (orange) mit studentsche t-Verteilung (blau)

Graphisch gesehen passt die t-Verteilung schon mal viel besser als die Normalverteilung. Wir führen desto trotz die Test-Statistiken durch und schauen, ob die Annahme der t-Verteilung getroffen werden kann.

Test-Statistik	p-Wert
Anderson-Darling	0.936139
Cramér-von Mises	0.943746
Kolmogorov-Smirnov	0.921394
Kuiper	0.876235
Pearson $\mathcal{X}^2$	0.962986
Watson U	0.865916

Tabelle 2: Test-Statistiken zur Überprüfung der Annahme von t-verteilten log-Renditen

Auch die Test-Statistiken bestätigen, dass die Annahme der t-verteilten log-Renditen nicht verworfen werden kann.

Für den *Mathematica* Code kopieren wir das erste Modell und tauschen die normalverteilten log-Renditen mit unseren *gefitteten* t-verteilten log-Renditen, welche wir in der Variable `Fit1` gespeichert haben. Der Befehl für das zweite Modell ergibt sich dann zu:

```
Modell2[ini_, tmax_, volatility_, muT_] :=
Module[{vol = volatility, mu = muT},
r = RandomVariate[Fit1, tmax + 1];
Table[ini*Exp[Sum[r[[j]], {j, 1, i}]], {i, 1, tmax}]]
```

Mit den selben Eingabewerten wie im ersten Modell kann man sich nun Aktienkursverläufe simulieren.

```
plotModell2 =
ListLinePlot[
Modell2[Last[KursTage17], Length[KursTage18], sigmaTage17,
muTage17], PlotStyle -> RandomColor[]]

Show[plotModell2, ListLinePlot[KursTage18], PlotRange -> All]

Out:=
```

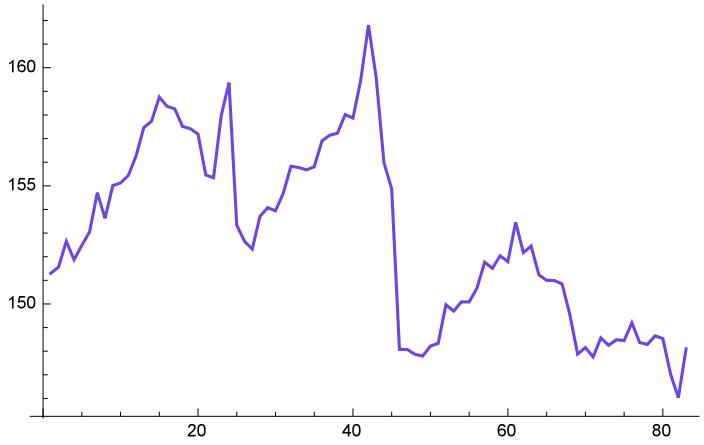

Abbildung 8: Simulierter Pfad mit Modul `Modell2`

Ob das zweite Modell nun besser ist als das erste, werden wir wie schon erwähnt in Kapitel 4, dem Ergebnisteil, diskutieren.

Im nächsten Abschnitt wenden wir uns einen anderen Ansatz zu. Dabei geht es um die geometrische brownsche Bewegung. In der berühmten Black Scholes Formel zur Bewertung von Finanzoptionen wird die Annahme getroffen, dass der Aktienpreis einer geometrischen brownschen Bewegung folgt (vgl. [3, Seite 272]). Im Folgenden wollen wir die Definition der geometrischen brownschen Bewegung vorstellen. Für die Herleitung verweisen wir auf das soeben zitierte Standardwerk von Hull.

2.3 M3: Geometrische brownsche Bewegung

Definition 2.2 (Geometrisch Brownsche Bewegung). Es sei W_t ein Wiener Prozess und S_t der Aktienkurs zum Zeitpunkt $t \in \mathbb{N}_0$. Ferner seien μ und σ Konstanten. Dann ist durch

$$S_t = S_0 \cdot \exp[(\mu - 0.5\sigma^2)t + \sigma W_t \sqrt{t}] \quad \text{mit} \quad W_t \sim N(0,1) \tag{2}$$

die geometrische Brownsche Bewegung gegeben (vgl. [1, Seite 6]). Wir nennen μ den Drift-Parameter und σ den Volatilitäts-Parameter. Der Mittelwert ist $E[S_t] = S_0 \cdot \exp[\mu t]$.

Dieses Prozess haben wir im selben Schema wie die anderen Prozesse selber programmiert:

```
Modell3[ini_, tmax_, volatility_, muT_] :=
Module[{vol = volatility, mu = muT},
Table[ini*
Exp[(mu - vol^2/2)*i +
vol*RandomVariate[NormalDistribution[]]*Sqrt[i]], {i, 1, tmax}]]
```

Alternativ bietet die Software *Mathematica* die Möglichkeit einen schon implementierten Befehl zu nutzen, `GeometricBrownianMotionProcess`. Die Ergebnisse unseres Moduls müssten relativ ähnlich zum implementierten Modul sein. In der folgenden Graphik wollen wir unser Modul mit dem vom *Mathematica* implementierten Modul und die Mittelwertsfunktion $E[S_t] = S_0 \cdot \exp[\mu t]$ vergleichen.

```
Mod3 = ListLinePlot[Mean[Table[Modell3[Last[KursTage17], 50, sigmaTage17, muTage17],
100000]], PlotStyle -> Black];

Mittelwertsfunktion = ListLinePlot[Table[Exp[t*muTage17]*Last[KursTage17], {t, 0, 50}],
PlotStyle -> Green];

GeoBroMot = ListLinePlot[Mean[Table[RandomFunction[GeometricBrownianMotionProcess[
muTage17,sigmaTage17, Last[KursTage17]], {0, 50, 1}]["Path"], {100000}]], PlotStyle -> Red];

Show[GeoBroMot, Mittelwertsfunktion, Mod3, PlotRange -> All]

Out:=
```

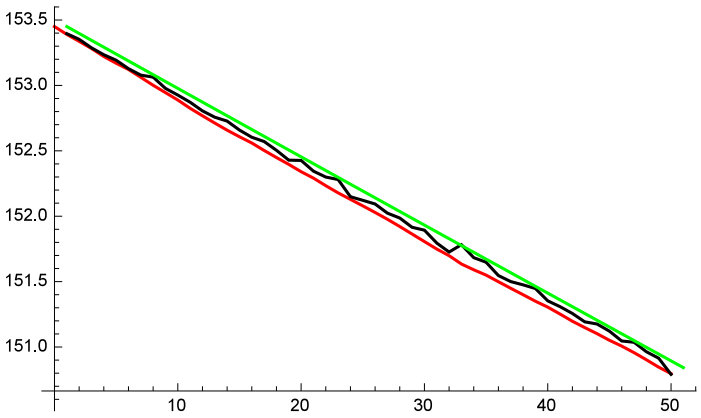

Abbildung 9: Vergleich der Verläufe `Modell3` (schwarz), GeometricBrownianMotion (rot), Mittelwertsfunktion (grün)

Es ist zu erkennen, dass die Werteverläufe sich nicht allzu sehr unterscheiden.

Im folgenden Abschnitt werden wir die ARIMA Modelle (**A**uto**R**egressive **I**ntegrated **M**oving **A**verage) kennenlernen, welche fester Bestandteil der modernen Zeitreihenanalyse sind.

2.4 M4: ARIMA Modell

Bevor wir die ARIMA Modelle einführen, wollen wir die ARMA Modelle definieren. Dabei werden wir diese Modelle nicht herleiten oder näher erläutern. Dazu verweisen wir auf das Buch von Shumway und Stoffer (vgl. [4]). Sämtliche Definitionen stammen aus diesem Werk.

Definition 2.3 (ARMA Modell). Der folgende stochastische Prozess X_t heißt ARMA(p,q)-Prozess:

$$X_t - \sum_{i=1}^{p} \phi_i X_{t-i} = \phi_0 + \sum_{i=1}^{q} \theta_i Z_{t-i} + Z_t \qquad Z_t \sim \mathrm{WN}(0, \sigma^2) \tag{3}$$

mit $\phi_0 = \mu(1 - \sum_{i=1}^{p} \phi_i)$. Eine alternative Schreibweise zu (3) ist folgende:

$$a(L)X_t = \phi_0 + b(L)Z_t \tag{4}$$

wobei L der Lag-Operator ist und wie folgt definiert ist:

$$Lr_t = r_{t-1}, \qquad L^j r_t = r_{t-j}$$

Definition 2.4 (ARIMA Modell). Der folgende stochastische Prozess X_t heißt ARIMA(p,d,q)-Prozess:

$$a(L)(1-L)^d X_t = \phi_0 + b(L)Z_t \tag{5}$$

In *Mathematica* sind die ARIMA Prozesse vordefiniert. Da die Variablen p, q und d zu wählen sind, stellt sich die Frage, wie diese ausgesucht werden sollen bzw. welcher dieser Prozesse am besten zu unserer Zeitreihe passt.

Mathematica nimmt uns mit dem Befehl `TimeSeriesModelFit` diese Arbeit ab. Es werden intern eine Reihe von Modellen mittels Akaikes Informationskriterium (AIC (vgl. [7])) verglichen und dabei wir das Beste ausgegeben. Eine Prognose für unsere Kursdaten mit einem ARIMA Modell sieht dann folgendermaßen aus:

```
ARIMA[EchtDaten_, tmax_] := Module[{}, TimeSeriesForecast[
TimeSeriesModelFit[EchtDaten], {Length[KursTage18]}][["Path"]][[All, 2]]];
```

```
Show[ListLinePlot[ARIMA[KursTage17, Length[KursTage18]],
PlotStyle -> Green], ListLinePlot[KursTage18], PlotRange -> All]

Out:=
```

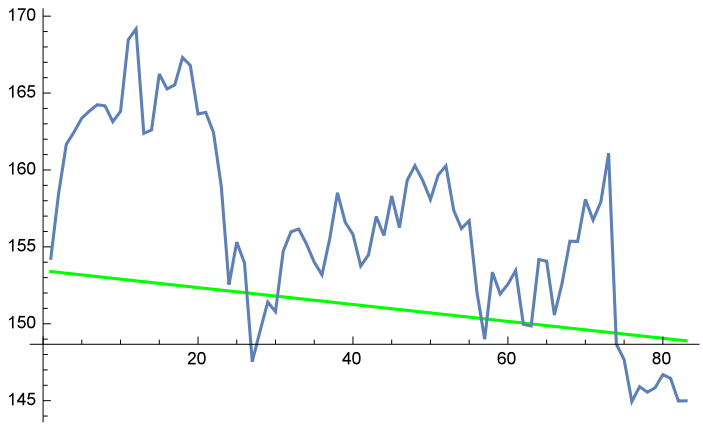

Abbildung 10: Vergleich der ARIMA Prognose (grün) mit dem tatsächlichem Verlauf (blau)

2.5 M5.1: Künstliche neuronale Netze in Python

Die künstlichen neuronalen Netze (KNN) (auch: artificial neural networks (ANN)) sind informationsverarbeitende Systeme, die eine Analogie zum menschlichen Gehirn haben. Die neuronalen Netze bestehen aus einer großen Anzahl an Neuronen, die sich Informationen in Form einer Aktivierung der Neuronen über gerichtete und gewichtete Verbindungen zusenden. Dabei haben diese die Fähigkeit eine Aufgabe eigenständig, nur anhand von Trainingsbeispielen, zu lernen. Die neuronalen Netze haben sehr unterschiedliche Einsatzmöglichkeiten, unter Anderem im Bereich der Prognosen und Kategorisierungen (vgl. [1]).

Die Funktionsweise und der Aufbau künstlicher neuronaler Netze wurde im Rahmen der *Data Mining 2* Vorlesung von *Prof. Dr. Werner Helm* (Hochschule Darmstadt) ausführlich vermittelt. Daher bezieht sich der folgende Abschnitt hauptsächlich auf die Vorlesungsunterlagen von Herr Helm und dem Artikel vom Herrn Balzer (vgl. [5]). Falls andere Quellen verwendet wurden, werden diese an den entsprechenden Stellen vermerkt.

Bei den künstlichen neuronalen Netzen wurde das menschliche Gehirn bzw. die Funktionsweise des Gehirns als Vorbild genommen. In der folgenden Abbildung erkennt man auf der linken Seite den Aufbau einer Nervenzelle und auf der rechten Seite die mathematische Darstellung für neuronale Netze.

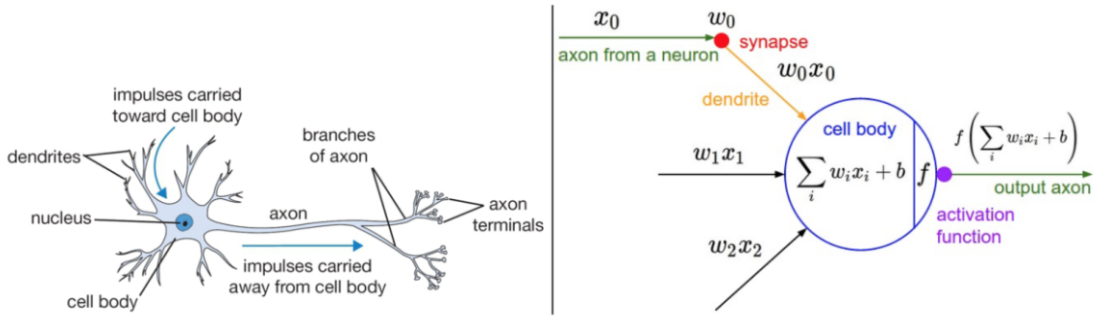

Abbildung 11: Aufbau einer Nervenzelle (links) und mathematische Darstellung für neuronale Netze (rechts)

Dabei besteht ein Neuron aus Zellkörpern, einen oder mehreren Eingängen und aus einem einzigen Ausgang (Axon), welcher über Synapsen verstärkende oder hemmende Signale (bzw. positive oder negative Gewichte) erhält. Im Neuron wird eine gewichtete Summe der Signale berechnet und sobald diese einen Schwellwert überschreitet, wird dementsprechend ein Ausgangssignal (oder auch Reiz genannt) an abgehende Axone geliefert. Je nachdem wie stark das Axon aktiviert ist, werden mehr oder weniger Reizsignale durchgelassen.

Auf der rechten Seite der Abbildung 11 bzw. auch in der nächsten Abbildung 12 sieht man das mathematische Neuronen-Modell. Hier werden die Eingangswerte $x_0, x_1, x_2, \ldots$ mit Gewichten $w_0, w_1, w_2, \ldots$ multipliziert, wobei es nach der Effizienz der einzelnen Input-Kanäle gewichtet und zum Schluss aufsummiert wird. Es wird noch ein Offset b hinzu addiert und zuletzt als Eingang in die Aktivierungsfunktion f übertragen.

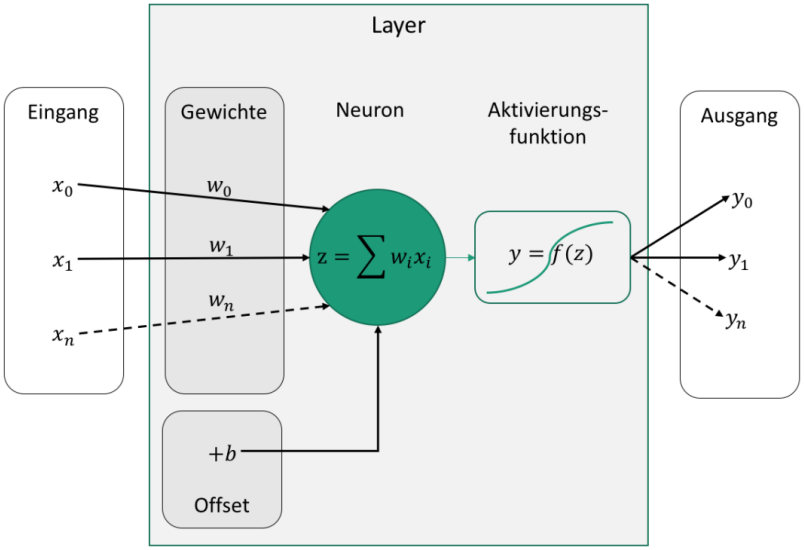

Abbildung 12: Darstellung des mathematischen Neuronen-Modells

Folgende Aktivierungsfunktionen werden am häufigsten eingesetzt, wobei natürlich noch weitere ver-

schiedene Aktivierungsfunktionen existieren:

- Identitätsfunktion: $f(x) = x$
- Hyperbolische Tangentenfunktion: $f(x) = tanh(x)$
- Sigmoidfunktion: $f(x) = \frac{1}{1+e^{-x}}$
- Schwellwertfunktion: $f(x) = 0$ wenn $x < 0$, 1 sonst
- ReLu (Rectifier Linear Unit) Funktion: $f(x) = \max(0, x)$

In der folgenden Abbildung 13 ist ein Ausschnitt von am meisten benutzten Aktivierungsfunktionen:

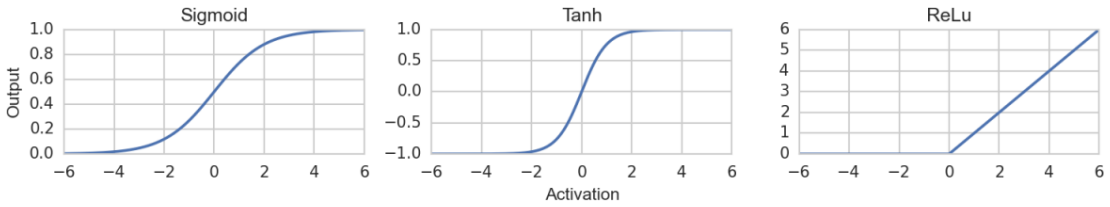

Abbildung 13: Darstellung der drei gebräuchlichsten Aktivierungsfunktionen

Wie soeben schon erläutert, gibt ein Neuron je nach Aktivierungsstärke ein Signal unterschiedlich stark weiter. Wobei hier noch beachtet werden muss, dass zu Beginn des Lernprozesses jedes Neuron zunächst initialisiert werden muss. Die Startgewichte dürfen nicht gleich groß gewählt werden, da das Backpropagation-Lernverfahren (später diesbezüglich mehr) nicht effizient genug arbeiten kann. Es werden kleine und zufällige Werte bei der Initialisierung des Netzes verwendet, wobei die Initialwerte der Gewichte oft im Bereich von -1 bis 1 liegen. Somit lässt ein Neuron je nach Aktivierungsfunktion ein Signal gar nicht (0) oder teilweise (-1) oder stark (+1) durch.

Zusammenfassend lässt sich sagen, dass die künstlichen neuronalen Netze schichtweise in sogenannten Layern (Schichten) angeordnet und in einer festen Hierarchie miteinander verbunden sind. Mit dem *Input Layer* fließen Informationen über eine oder mehrere *Hidden Layer* bis hin zum *Output Layer*. Natürlich kann auch der Output eines Neurons der Input des anderen Neurons sein (vgl. [8]).

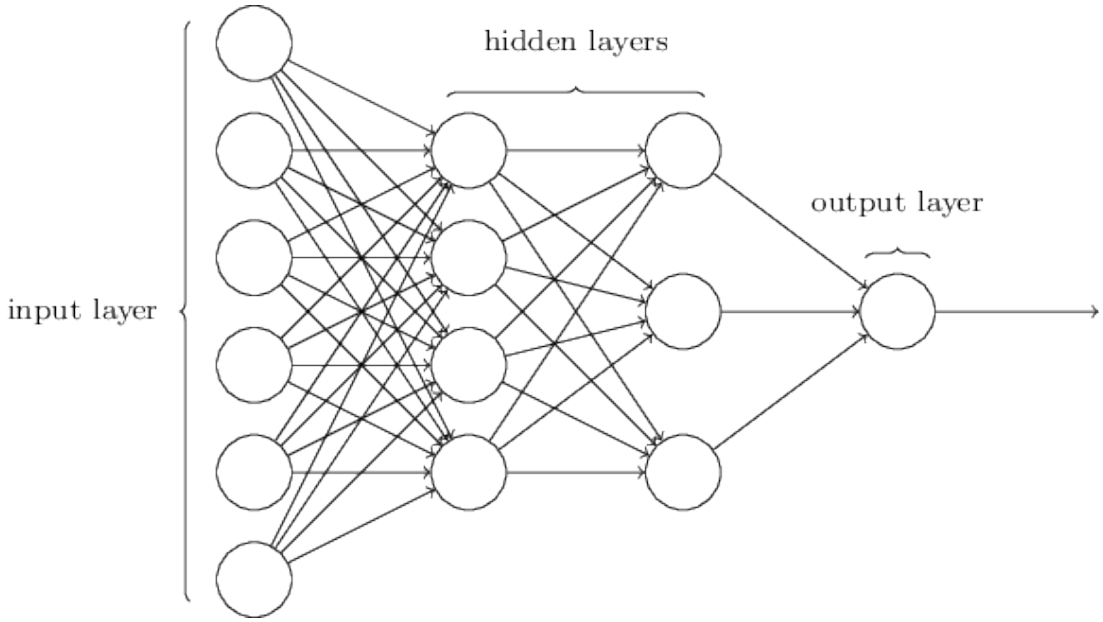

Abbildung 14: Schematische Darstellung eines einfachen neuronalen Netzes

Es werden im Folgenden die drei am häufigsten verwendeten Arten der neuronalen Netze kurz beschrieben, welche später eine hohe Relevanz für das Verständnis der Vorhersage haben werden. Zu diesen zählen die *Feedforward Netze*, *Rekurrente Netze* und *Long short-term memory Netze*.

Bei *Feedforward Netzen (FF)* werden Informationen vom Input Layer über die Hidden Layer bis hin zum Output Layer in eine Richtung „forward“ (also vorwärts) weitergeleitet.

Rekurrente Netze (RNN), auch rückgekoppelte oder Feedback Netze genannt, ähneln den Feedforward-Netzen. Es existieren hierbei jedoch zusätzliche Verbindungen durch die Informationen bestimmte Bereiche des Netzwerkes auch rückwärts beziehungsweise erneut durchlaufen können.

Long short-term memory (LSTM) (auf deutsch: langes Kurzzeitgedächtnis) sind angepasste RNN, welche aber so konstruiert sind, dass diese sich Informationen über einen größeren Zeitraum merken können. RNN hat ein sogenanntes *vanishing exploding gradients* Problem, welches durch LSTM mit Hilfe von input and forget gates gelöst wird (vgl. [8]).

In der folgenden Graphik 15 befindet sich eine übersichtliche Darstellung einiger Versionen neuronaler Netze, die im Laufe dieser Ausarbeitung als Modelle verwendet werden. Die gesamte Überblick kann auf der Webseite (vgl. [8]) betrachtet werden.

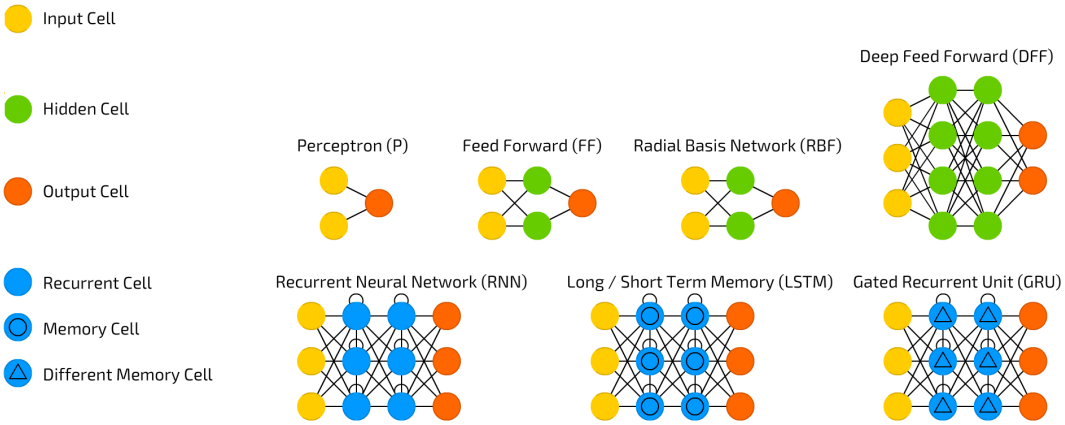

Abbildung 15: Überblick verschiedener Versionen neuronaler Netze mit deren Aufbau

Nach der Topologie neuronaler Netze werden diese nun in drei Phasen eingeteilt: Lern-, Test- und Anwendungsphase. In der Anwendungsphase wird das zuvor antrainierte Netz zur Lösung des spezifischen Problems (hier: Vorhersage der IBM Aktienkursverläufe) eingesetzt. Zuvor müssen die Lern- und Testphasen erläutert werden. Bevor es zum Lernen des Netzes geht, muss man wissen ob die Vorhersage, also der Output, richtig oder falsch klassifiziert wurde. Hierbei sollte noch beachtet werden, dass die Netze weder Informationen über interne Zusammenhänge oder jegliche statistische Modelle benötigen. Denn bei der Lernphase werden die Verbindungsgewichte modifiziert und dadurch erzeugen die Netze selbstständig Informationen über die Struktur anhand der Beispieldaten bzw. Trainingsdaten.

Wie auch schon in der Einleitung erwähnt, sind die neuronale Netze massiv lernfähige Systeme, die anhand von Trainingsbeispielen lernen. Beim Training entsteht eine Differenz zwischen der tatsächlichen Ausgabe des neuronalen Netzes und einer gewünschten Ausgabe, welche in den Trainingsdaten bekannt ist. Somit entstehen also Trainings- oder auch Netzwerkfehler, wobei man am Ende diesen Fehler natürlich minimieren möchte. Je nach Anwendungsfall werden verschiedene Zielfunktionen bzw. Loss- oder Fehlerfunktionen definiert. Beim Klassifizierungsproblem würde man beispielsweise die korrekt erkannten Fälle maximieren wollen und bei der Vorhersage eines Wertes würde man den mittleren quadratischen Fehler minimieren wollen. Daher ist es wichtig die Zielfunktion auf den entsprechenden Anwendungsfall anzupassen (vgl. [1]).

Die Lernverfahren zum Training der künstlichen neuronalen Netze werden in drei grundsätzliche Kategorien eingeteilt: überwacht, bestärkt oder unüberwacht (vgl. [1]).

- *Überwachtes Lernen (supervised learning):*
  Hier werden zu allen Eingabedaten auch die dazugehörige Ausgabedaten angegeben. Die Aufgabe des Netzes ist, die Verbindungsgewichte so anzupassen bzw. optimieren, dass die Ausgabe gut approximiert wird.
- *Bestärkendes Lernen (reinforcement learning):*
  Es wird nur angegeben, ob die Ausgabe richtig oder falsch war. Somit erfährt das Netz nicht den Wert des Unterschieds und muss anhand dieser Information die korrekte Ausgabe selbständig herausfinden.
- *Unüberwachtes Lernen (unsupervised learning):*
  Beim Lernen werden dem Netz ausschließlich Eingabedaten und keine Ausgabedaten vorgegeben. Das Netz kann nun eigenständige Regeln aufstellen und versucht ähnliche Eingaben in ähnliche Kategorien zu klassifizieren.

In dieser Ausarbeitung wird speziell auf das überwachte Lernen eingegangen. Es entsteht hierbei ein Fehler zwischen der gewünschten und der tatsächlichen Ausgabe. Beim überwachten Lernen wird bei der Modifikation der Gewichte der Backpropagation-Algorithmus verwendet. Die folgende Fehlerfunktion

$$E = \frac{1}{2} \sum_{i=1}^{n} (t_i - o_i)^2$$

beschreibt den quadratischen Fehler mit n als Anzahl der Trainingsdaten, t_i als target Wert (Zielausgabe) und o_i als output Wert (vorhergesagte Ausgabe). Das Ziel ist es, die Fehler mittels einem Verfahren zur unrestringierten (ohne Nebenbedingung) Optimierung zu minimieren. Es existieren viele Verfahren wie beispielsweise: Stochastic Gradient Descent (SGD), Nesterov Momentum, Adagrad und Adadelta oder Rmsprop. Zur Bearbeitung dieser Aufgabe haben wir uns auf ADAM beschränkt, welches im Moment einer der am häufgsten angewendeten Optimierer ist. ADAM steht für Adaptive moment estimation und ist eine methodische Kombination von den Optimierungstechniken AdaGrad und RMSProp. Der Fehler, den das Netzwerk bei einer Schätzung des Ausgangswertes macht, wird über Backpropagation auf die jeweiligen Neuronen zurück verteilt, welche ihn auch verursacht haben. Die Gewichte sollen dabei solange angepasst werden, bis der Fehler zwischen der Zielausgabe und der Netzausgabe am geringsten ist.

Nach der Lernphase findet die Anwendungsphase statt. Dabei generieren die Netze eine Netzausgabe. Hier werden die Verbindungsgewichte, die durch die Lernphase ermittelt und fixiert wurden, an die neuen Daten angewendet und die neuronalen Netze treffen Aussagen über den Output bzw. erstellen diese Prognosen.

2.5.1 Feedforward und LSTM Netze für Log Returns

Im folgenden wollen wir einführend die Prognose der Log Returns durchführen, bevor wir die Aktienschlusskurse prognostizieren. Anschließend werden wir mit den neuronalen Netzen in Python auch Kurzzeitprognosen durchführen und die Ergebnisse analysieren. Für die Prognose verwenden wir Open-high-low-close Kurse, sodass unser neuronales Netz mehr Informationen zur Verfügung hat. Dabei erhoffen wir uns eine viel bessere Prognose als mit den vorigen Modellen.

Für die Prognose der IBM Log Returns lesen wir die Open-high-low-close Kurse (OHLC-Kurse) in Python ein und bereinigen diese. Das heißt, der Datensatz muss für die spätere Anwendung vorbereitet werden, falls fehlende Werte auftauchen sollten. Zudem werden am Anfang die für die Erstellung der neuronalen Netze benötigte Pakete importiert. Um das neuronale Netz zu konstruieren, braucht man zum einen das Paket TensorFlow und zum anderen Keras. Mithilfe von TensorFlow findet der Lernprozess, bei dem das neuronale Netz optimiert wird, statt und Keras benötigt diese Bibliothek als Backend. Mittels Keras werden die Modelle definiert und anschließend an TensorFlow übergeben.

```
import numpy as np
import pandas as pd
import talib
import tensorflow
import keras
import random

random.seed(1234)
dataset = pd.read_csv('C:\\BUESRA\\Uni\\Master\\Pflichtfaecher\\Projekt
                       Simulation SS 18\\DATEN IBM\\ohlc_jan17_feb18.csv')
dataset = dataset.dropna()
dataset = dataset[['<OPEN>', '<HIGH>', '<LOW>', '<CLOSE>']]
dataset.head()
```

	<OPEN>	<HIGH>	<LOW>	<CLOSE>
0	167.130	167.86	166.02	167.18
1	167.615	169.84	167.42	169.24
2	169.250	169.36	167.29	168.72
3	168.800	169.91	167.66	169.56
4	169.470	169.73	167.66	167.71

Tabelle 3: Python Output der OHLC-Kurse

Im Folgenden können die logarithmierte Renditen für den gesamten Zeitraum ab Januar 2017 betrachtet werden.

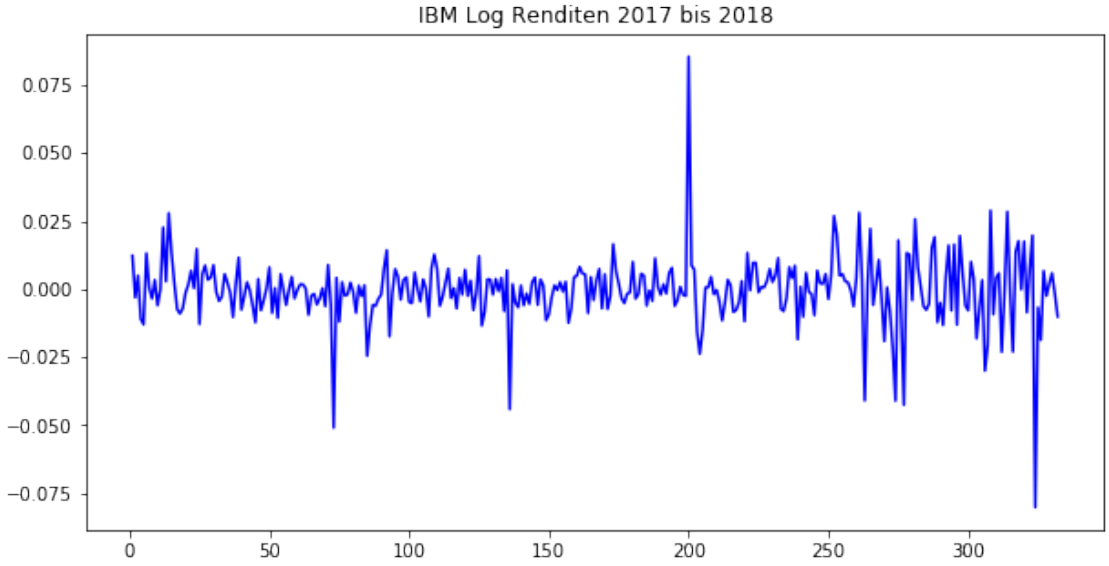

Abbildung 16: Verlauf der logarithmierten IBM Renditen ab dem Jahr 2017

Mit Hilfe der Informationen über Open-, High-, Low- und Close-Kurs können nun unabhängige Variablen erstellt werden. Hier werden zwei von drei neuen Variablen durch einfache Differenz des höchsten und niedrigsten Tageskurs und zwischen Eröffnungs- und Tagesschlusskurs definiert. Die anderen beiden Variablen sind die logarithmierten Renditen. Hier können natürlich noch weitere sinnvolle Variablen definiert und in das Modell mit rein genommen werden. Die Vorhersage, hier die Log Rendite zum Zeitpunkt $t+1$, soll anhand der drei unabhängigen Variablen getroffen werden.

```
dataset['H-L'] = dataset['<HIGH>'] - dataset['<LOW>']
dataset['O-C'] = dataset['<CLOSE>'] - dataset['<OPEN>']
dataset['Log Renditen'] = np.log(dataset['<CLOSE>']/dataset['<CLOSE>'].
shift(1))
dataset['Log Renditen(t+1)']=dataset['Log Renditen'].shift(-1)

dataset = dataset.dropna()
dataset = dataset.dropna(thresh=2)
dataset.head()
```

Als Ausgabe erhalten wir eine Tabelle folgender Form:

	<OPEN>	<HIGH>	<LOW>	<CLOSE>	H-L	O-C	Log Renditen	Log Renditen(t+1)
1	167.615	169.84	167.42	169.24	2.42	1.625	0.012247	-0.003077
2	169.250	169.36	167.29	168.72	2.07	-0.530	-0.003077	0.004966
3	168.800	169.91	167.66	169.56	2.25	0.760	0.004966	-0.010971
4	169.470	169.73	167.66	167.71	2.07	-1.760	-0.010971	-0.012963
5	167.960	167.99	165.42	165.55	2.57	-2.410	-0.012963	0.013142

Tabelle 4: Python Output der prognostizierten logarithmierten Renditen

Nun werden die neu definierten Variablen für die weitere Ausarbeitung als X und Y bezeichnet.

```
X = dataset[['<CLOSE>', 'H-L','O-C']]
y = dataset[['Log Renditen(t+1)']]
```

Für X erhalten wir die Daten in folgender Form:

```
X.head()
```

	<CLOSE>	H-L	O-C
1	169.24	2.42	1.625
2	168.72	2.07	-0.530
3	169.56	2.25	0.760
4	167.71	2.07	-1.760
5	165.55	2.57	-2.410

Tabelle 5: Python Output unserer Datenmenge X

und für y erhalten wir die Daten analog in folgender Form:

```
y.head()
```

	Log Renditen(t+1)
1	-0.003077
2	0.004966
3	-0.010971
4	-0.012963
5	0.013142

Tabelle 6: Python Output unserer Datenmenge y

Im Rahmen der Datenaufbereitung werden die Daten zusätzlich einheitlich skaliert. Zudem wird der Datensatz manuell in Trainings- und Testmengen aufgeteilt, d.h. die Beobachtungen aus dem Jahre 2017 werden als Trainingsmenge verwendet und die Beobachtungen ab Anfang 2018 bis Anfang Mai als Testmenge.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
X = scaler.fit_transform(X)
y = scaler.fit_transform(y)

X_train = X[:251]
X_test = X[251:]
y_train = y[:251]
y_test = y[251:]
```

Das Ziel ist es nun, die Renditen für das Jahr 2018 zu prognostizieren, welche in der folgenden Abbildung 17 zu entnehmen sind.

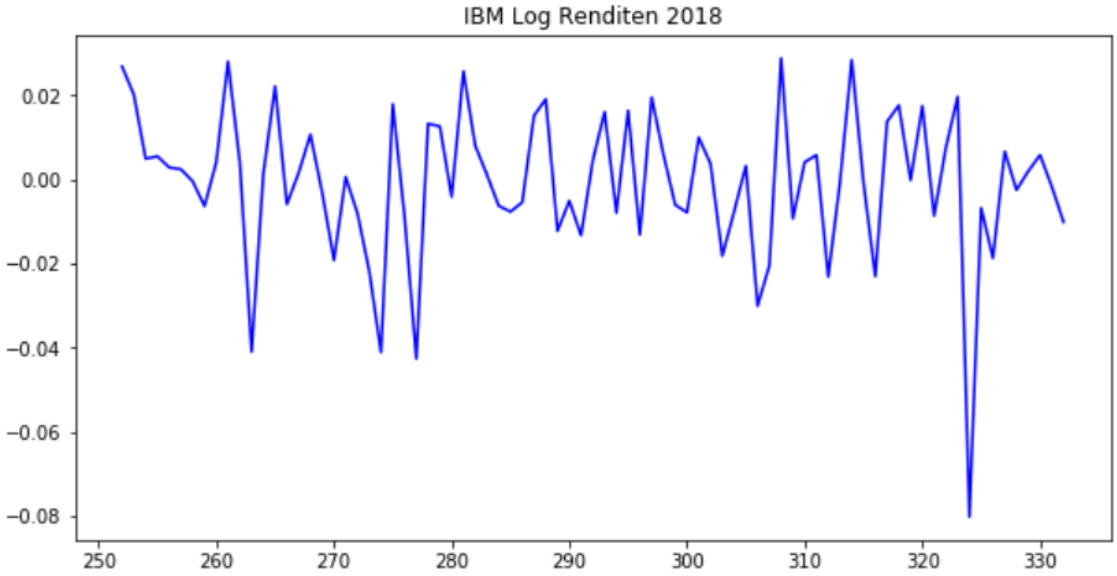

Abbildung 17: Verlauf der logarithmierten IBM Renditen ab dem Jahr 2018

Nun wollen wir das neuronale Netz konstruieren. Die Keras Python-Bibliothek für Deep Learning konzentriert sich auf die Erstellung von Modellen als eine Reihe von Schichten (sequence of layers). Daher wird im Folgenden mit den Sequential aus der Keras Bibliothek gearbeitet. Mit der Funktion Sequential() baut man ein Modell, mit der man manuell Schichten zum neuronalen Netz hinzufügen kann. Dabei wird die Methode add() verwendet, in der man die Art der Schicht definiert. Mit Dense() fügt man eine reguläre Schicht hinzu, bei der jeder Knoten dieser Schicht mit jedem Knoten der nächsten Schicht verbunden ist. Nur bei der ersten Schicht wird die Anzahl der Inputvariablen notwendig eingetragen (hier 4 Regressoren, also 4 inputs - kann mit X.shape eingetragen werden). Zudem wird innerhalb dieser Funktion auch die Anzahl der Knoten (hier: 128) definiert. Mit dem Argument kernel_initializer wird der Anfangszustand aller Gewichtungen zwischen den Knoten, welche während des Trainings optimiert werden, bestimmt. Die Initialisierung kann im einfachsten Fall gleichverteilt, also uniform, erfolgen. Danach wird noch mit Hilfe des Arguments activation zuletzt bestimmt, wie jeder Knoten der Schicht auf den jeweiligen Input reagieren soll. Es werden im Folgenden die Aktivierunsfunktionen relu und linear verwendet.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout

knn = Sequential()

knn.add(Dense(units = 12, kernel_initializer = 'uniform',
activation = 'relu', input_dim = X.shape[1]))

knn.add(Dense(units = 8, kernel_initializer = 'uniform',
activation = 'relu'))

knn.add(Dense(units = 1, kernel_initializer = 'uniform',
activation = 'linear'))
```

Es wird hier also ein neuronales Netz mit zwei versteckten Schichten (two hidden layers) gebaut. Hierbei sind alle Inputvariablen mit allen Knoten der versteckten Schicht verbunden. Bei der ersten versteckten Schicht gibt es 12 Knoten die mit 3 Inputvariablen verbunden werden. Diese hidden layer wird dann wieder mit einer anderen hidden layer mit 8 Knoten verbunden. Zu der letzte hidden layer fügt man nun

noch eine weitere Schicht hinzu. Sie entspricht dem Output und diese Schicht besitzt nur einen Knoten.

Im Folgenden sieht man die schematische Darstellung des neuronalen Netzes. Diese wurde mit Hilfe der IPython.display Bibliothek und dem dazugehörigen SVG Modul abgebildet.

```
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot

SVG(model_to_dot(knn, show_shapes=True, rankdir='LR').
create(prog='dot', format='svg'))
```

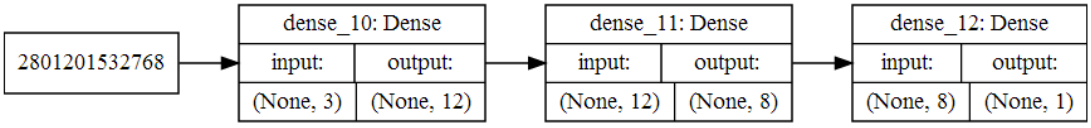

Abbildung 18: Schematische Darstellung unseres neuronalen Netzes (KNN))

Bevor das Modell trainiert wird, muss es erst kompiliert werden. Hier wird definiert wie der Lernprozess ablaufen soll. Es wird dem Algorithmus übergeben welche Fehlerfunktion auf welche Art und Weise minimiert werden soll und welche Kennzahl während der Optimierung ausgegeben werden soll. Danach kann das Modell auch schon trainiert werden.

Der Mean Squared Error ist eine häufig verwendete loss function im Bereich der neuronalen Netze. Das Ziel ist es, den Mean Squared Error möglichst nahe an 0 zu bringen. Dann ist der Fehler am geringsten und das neuronale Netz sagt die Prognose, also den Outcome, möglichst gut voraus. Die Neuronen wurden, wie oben schon erläutert, im ersten Schritt mit zufälligen Aktivierungen initialisiert. Das Netzwerk berechnet nun die Ausgabegröße und wird von der Zielfunktion (Loss) bestraft. Der Fehler den es gemacht hat, wird über Backpropagation auf die jeweiligen Neuronen zurück verteilt, die ihn verursacht haben.

Es wird geschaut ob der Fehler größer oder kleiner wird, wenn man die Aktivierung erhöht. Mathematisch gesehen wird also die Steigung der Fehlerfunktion bestimmt. Idealerweise gibt es eine Richtung in die man optimieren kann, sodass die Fehlerfunktion minimiert wird. Somit kommt man bei der Fehlerminimierung an. Die Suche nach der idealen Aktivierung bedeutet mathematisch das Finden von Minimalwerten im Fehlerraum. Es gibt viele Verfahren wie schon im theoretischen Teil über neuronale Netze erläutert wurde. Wir verwenden ADAM, welches im Moment einer der am häufigsten angewendeten Optimierer ist.

```
knn.compile(optimizer = 'adam', loss = 'mean_squared_error')

hist = knn.fit(X_train, y_train, batch_size = 10, epochs = 50,
    validation_data=(X_test, y_test), verbose=2, shuffle=False)
```

```
Train on 251 samples, validate on 81 samples
Epoch 1/50
 - 0s - loss: 0.2219 - val_loss: 0.2056
Epoch 2/50
 - 0s - loss: 0.1808 - val_loss: 0.1515
Epoch 3/50
 - 0s - loss: 0.1138 - val_loss: 0.0697
Epoch 4/50
 - 0s - loss: 0.0344 - val_loss: 0.0149
Epoch 5/50
 - 0s - loss: 0.0056 - val_loss: 0.0170
Epoch 6/50
 - 0s - loss: 0.0082 - val_loss: 0.0146
```

Abbildung 19: Auswertung der Fehlerfunktion im neuronalen Netz (KNN)

Die kompletten Daten werden insgesamt 50-mal durch das neuronale Netz geschoben. In jedem dieser Durchgänge werden die Daten in Batches von je 10 Instanzen in das Netz eingespeist. Nach jedem Batch wird eine Fehlerfunktion berechnet und die Gewichte zwischen den Knoten via Backpropagation entsprechend angepasst, so dass der Schätzfehler kleiner wird. Während der Algorithmus arbeitet, wird der Fortschritt vom Programm ausgegeben.

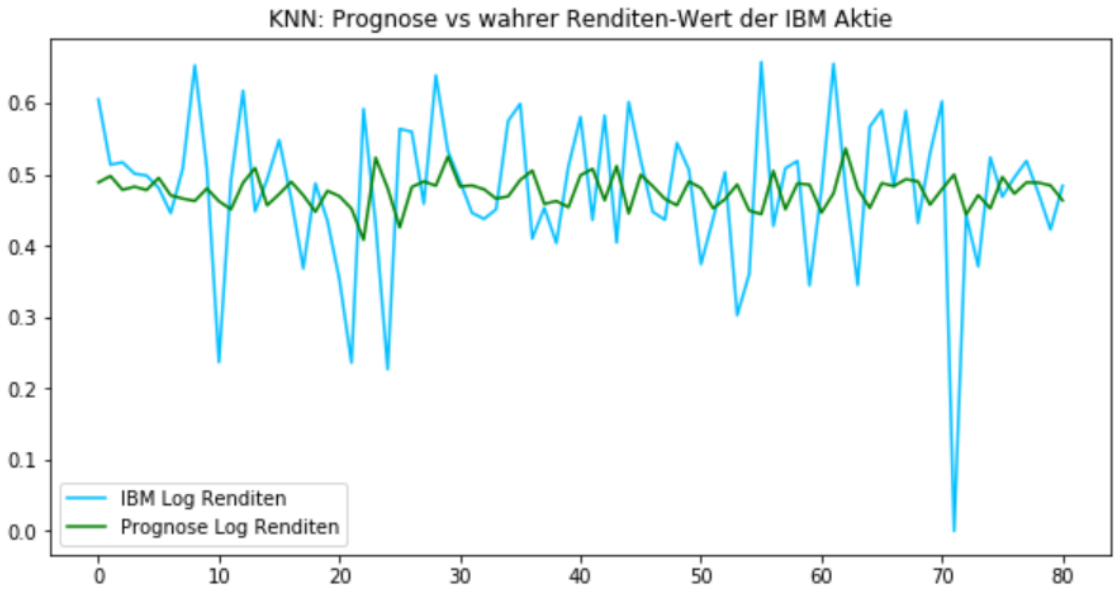

Abbildung 20: Tatsächlicher Verlauf der Renditen im Jahr 2018 und der prognostizierte Verlauf (KNN)

Mit Hilfe des sklearn.metrics Paket kann man zusätzlich Gütemaße wie MSE (Mean Squared Error) oder RMSE (Root Mean Squared Error) berechnen.

```
from math import sqrt
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_test, y_pred)
print('Test MSE: %.4f' % mse)

rmse = sqrt(mean_squared_error(y_test, y_pred))
print('Test RMSE: \%.4f' \% rmse)
```

Für den MSE erhalten wir einen Wert von 0.0125 und für den RMSE einen Wert von 0.1116.

Nun können nicht nur die Prognosewerte geplottet werden, sondern auch der Verlauf der Lossfunktion bzw. die Konvergenz der Lossfunktion. Anhand dessen kann auch beurteilt werden wie gut ein neuronales Netz arbeitet.

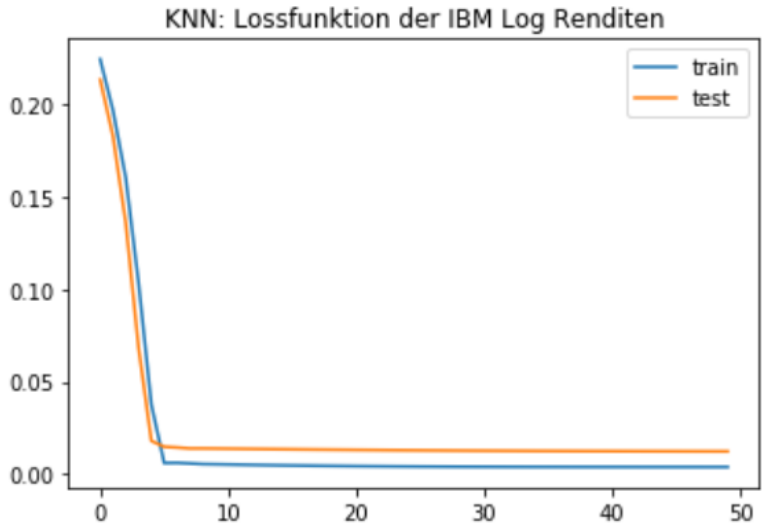

Abbildung 21: Plot der Loss-Funktion für das neuronale Netz (KNN)

Wir wollen an dieser Stelle noch anmerken, dass in dieser Ausarbeitung nur die skalierten Daten erläutert werden und somit auch die Gütemaße und Prognosen skaliert sind. Die inverse Skalierung der Daten und dazugehörige Prognosen und Plots können in der jeweiligen Python Datei im beigefügten Ordner entnommen werden.

Nun wird ein LSTM Netz für die Prognose der logarithmierten Renditen verwendet.

```
lstm = Sequential()

lstm.add(LSTM(64, input_shape=(X_train.shape[1], X_train.shape[2])))

lstm.add(Dense(1))

lstm.compile(loss='mean_squared_error', optimizer='adam')

history = lstm.fit(X_train, y_train, epochs=50, batch_size=72,
    validation_data=(X_test, y_test), verbose=2, shuffle=False)
```

```
Train on 252 samples, validate on 80 samples
Epoch 1/50
 - 1s - loss: 0.2268 - val_loss: 0.2102
Epoch 2/50
 - 0s - loss: 0.2016 - val_loss: 0.1859
Epoch 3/50
 - 0s - loss: 0.1781 - val_loss: 0.1632
Epoch 4/50
 - 0s - loss: 0.1561 - val_loss: 0.1423
Epoch 5/50
 - 0s - loss: 0.1358 - val_loss: 0.1231
Epoch 6/50
 - 0s - loss: 0.1171 - val_loss: 0.1054
```

Abbildung 22: Auswertung der Fehlerfunktion im neuronalen Netz (LSTM)

Die dazugehörige schematische Darstellung des neuronalen Netzes sieht wie folgt aus:

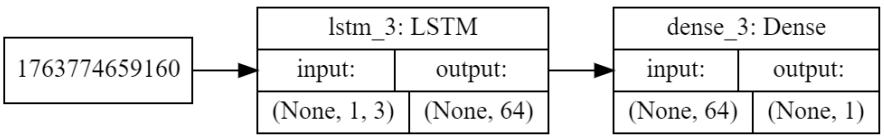

Abbildung 23: Schematische Darstellung unseres neuronalen Netzes (LSTM)

Bei der Prognose der LSTM Netze ist deutlich zu erkennen, dass nur die Spitzen der gestiegenen logarithmierten Renditen erreicht wird, aber nicht die fallenden Tiefpunkte.

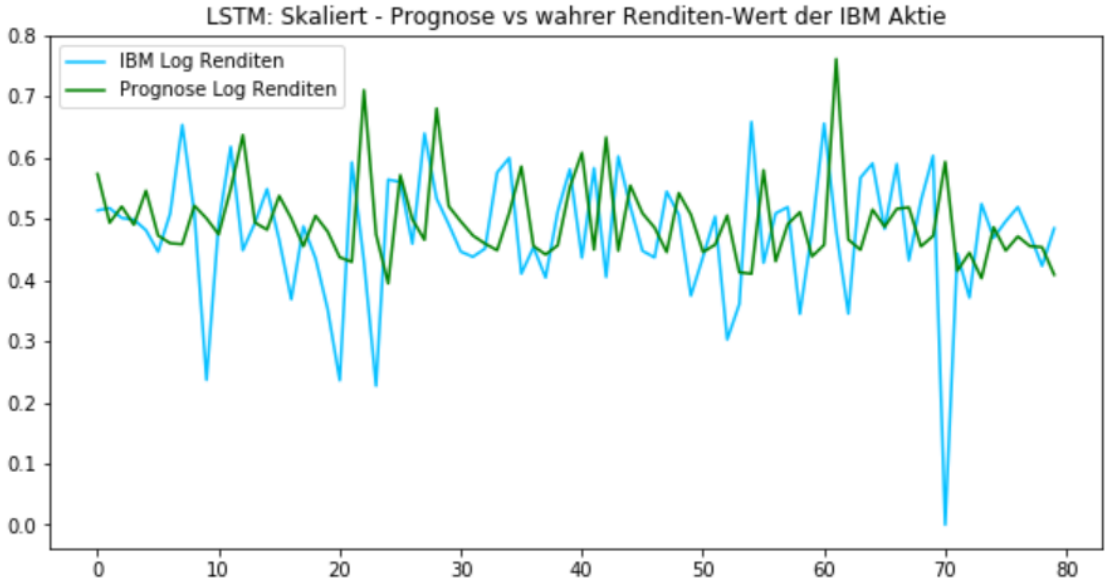

Abbildung 24: Tatsächlicher Verlauf der Renditen im Jahr 2018 und der prognostizierte Verlauf (LSTM)

Anhand des Abfalls der Lossfunktion und des MSE Wertes kann zwar gesagt werden, dass das Feedforward Netz besser ist, nichtsdestotrotz erkennt man beim Feedforward Netz, dass dieses im geringen

Maß die Steigung und Gefälle der Rendite vorhersagen kann. Zudem werden dort auch die Höhepunkte nicht annähernd erreicht.

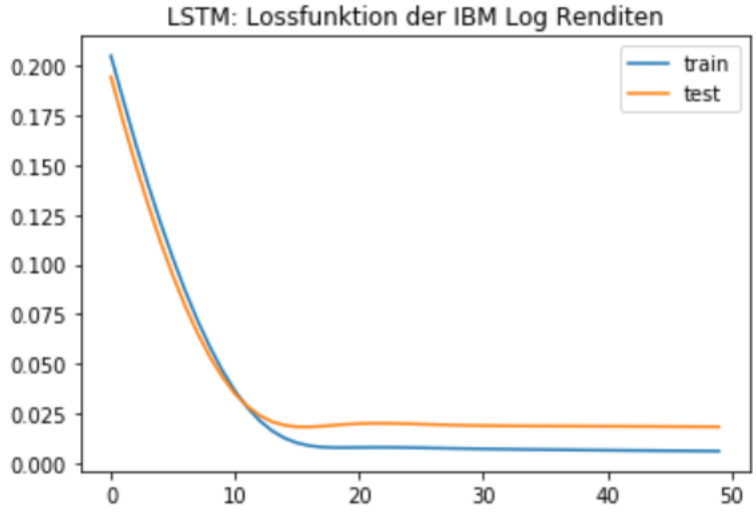

Abbildung 25: Plot der Loss-Funktion für das neuronale Netz (LSTM)

```
from math import sqrt
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_test, y_pre)
print('Test MSE: %.4f' % mse)

rmse = sqrt(mean_squared_error(y_test, y_pre))
print('Test RMSE: %.4f' % rmse)
```

Für den MSE erhalten wir einen Wert von 0.0183 und für den RMSE einen Wert von 0.1351.

2.5.2 Feedforward und LSTM Netze für Abschlusskurse

Bisher haben wir lediglich die logarithmierten Renditen der IBM Aktienkurse vorhergesagt. Nun wird ein anderer Ansatz gewählt. Es ist aus den Derivate 1 und Derivate 2 Vorlesungen bekannt, dass die Aktienanalyse mittels *log-return, Aktien Preis Differenz oder Aktienpreis bzw. Aktienschlusspreis* durchgeführt werden kann:

- Aktienpreis: s_n ist der Aktienschlusspreis am Tag n
- Preis Differenz: $x_n = s_n - s_{n-1}$
- Log Renditen: $r_n = \frac{s_n}{s_{n-1}}$

Nun werden im Folgenden Feedforward und LSTM Netze bei der Vorhersage der Aktienschlusspreis bzw. der Close Wert der Aktie verwendet.

Die abhängige Variable, und somit auch das was es hier zu prognostiziert gilt, ist der Aktienschlusskurs Wert am darauffolgenden Tag.

```
dataset['H-L'] = dataset['<HIGH>'] - dataset['<LOW>']
dataset['O-C'] = dataset['<CLOSE>'] - dataset['<OPEN>']
dataset['Close(t+1)']=dataset['<CLOSE>'].shift(-1)
```

```
dataset = dataset.dropna()
dataset = dataset.dropna(thresh=2)
dataset.head()
```

	<OPEN>	<HIGH>	<LOW>	<CLOSE>	H-L	O-C	Close(t+1)
0	167.130	167.86	166.02	167.18	1.84	0.050	169.24
1	167.615	169.84	167.42	169.24	2.42	1.625	168.72
2	169.250	169.36	167.29	168.72	2.07	-0.530	169.56
3	168.800	169.91	167.66	169.56	2.25	0.760	167.71
4	169.470	169.73	167.66	167.71	2.07	-1.760	165.55

Tabelle 7: Python Output der prognostizierten Schlusskurse

Anhand der folgenden Abbildung der Lossfunktion und der Prognose ist es erkennbar, dass die Feed-forward Netze bei der Testmenge ziemlich gute Vorhersagen liefern.

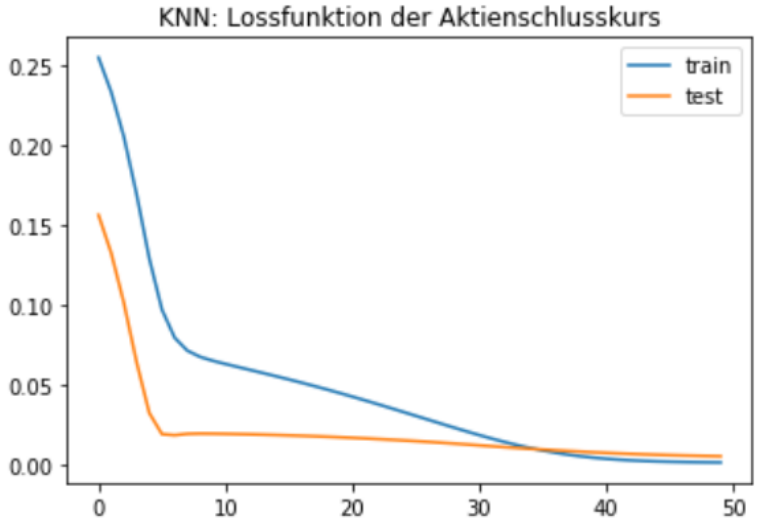

Abbildung 26: Plot der Loss-Funktion für das neuronale Netz 2 (KNN)

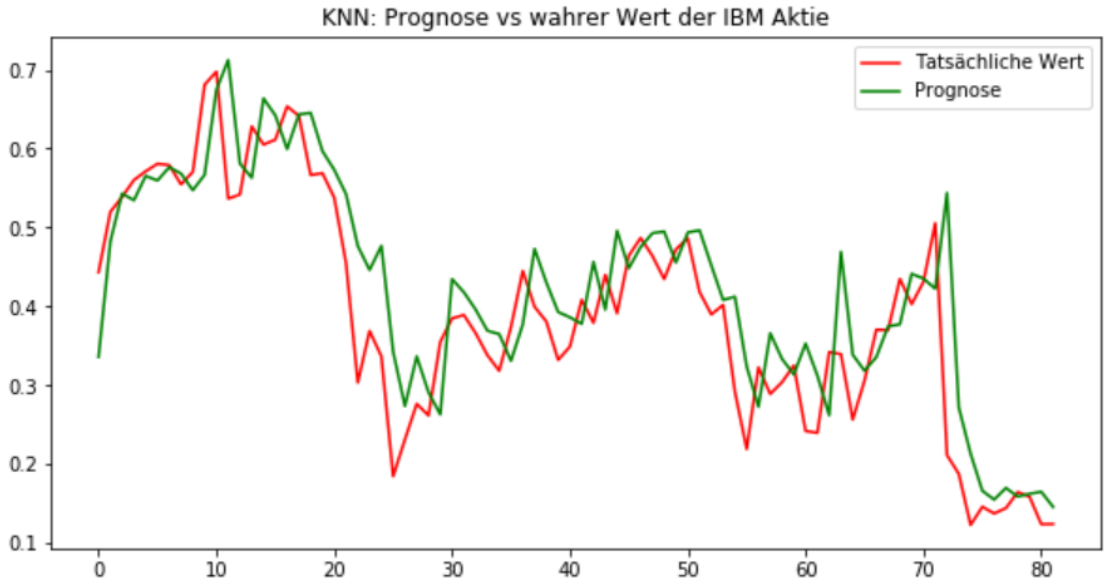

Abbildung 27: Tatsächlicher Verlauf der Schlusskurse im Jahr 2018 und der prognostizierte Verlauf (KNN)

Es ist nicht verwunderlich, dass der MSE und der RMSE sehr gute Werte liefern.

```
from math import sqrt
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_test, y_pred)
print('Test MSE: %.4f' % mse)

rmse = sqrt(mean_squared_error(y_test, y_pred))
print('Test RMSE: %.4f' % rmse)
```

Der MSE liefert dabei einen Wert von 0.0057 und der RMSE liefert den Wert 0.0757.

Nun wird das LSTM Netz für die Vorhersage der Aktienschlusskurse angewendet. Hier ist die Prognose, im Vergleich zu den Feedforward Netzen, deutlich schlechter was auch die Lossfunktion und auch der MSE Wert erkennbar macht.

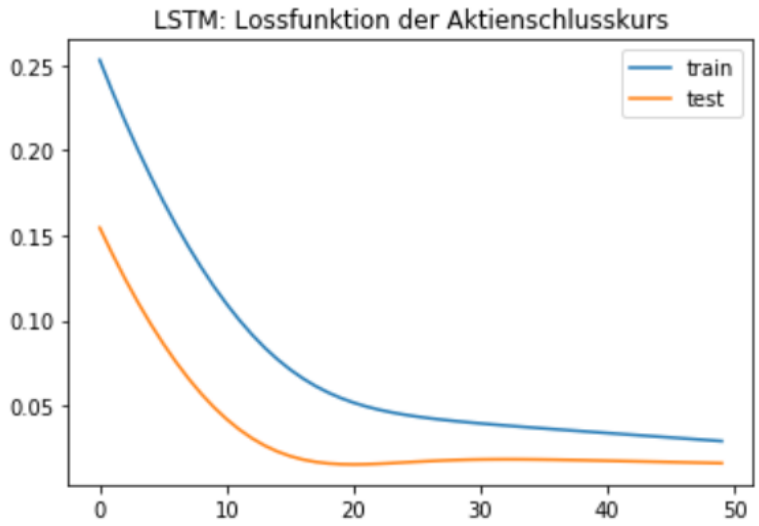

Abbildung 28: Plot der Loss-Funktion für das neuronale Netz 2 (LSTM)

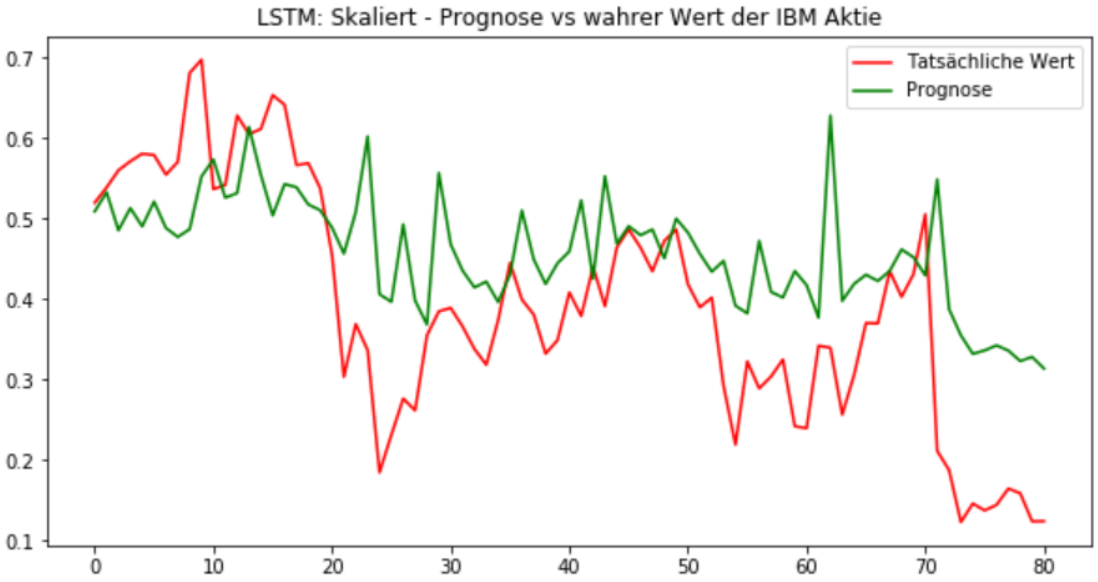

Abbildung 29: Tatsächlicher Verlauf der Schlusskurse im Jahr 2018 und der prognostizierte Verlauf (LSTM)

```
mse = mean_squared_error(y_test, y_pre)
print('Test MSE: %.4f' % mse)

rmse = sqrt(mean_squared_error(y_test, y_pre))
print('Test RMSE: %.4f' % rmse)
```

Der MSE liefert dabei einen Wert von 0.0163 und der RMSE liefert den Wert 0.1276, sodass das LSTM in diesem Fall schlechter ist als das Feedforward Netz.

2.5.3 Anwendung in Python - Kurzzeitprognose (KNN und LSTM)

Zusätzlich wurden Kurzzeitprognosen mit neuronalen Netzen, Feedforward und Long Short Term Memory Netzen, erstellt. Hierbei wurde auf einem bestimmten kurzen Zeitpunkt zum einem die logarithmierte Renditen und zum anderen die Schlusskurse der IBM Aktie vorhergesagt. Bei einem willkürlich gewählten Zeitpunkt werden die ersten sechs Tage als Trainings- und die nächsten zwei Tage als Testmenge verwendet. Als Beispieldatensatz werden die Tage ab 01. Februar 2018 gewählt.

Wir betrachten also folgenden Verlauf der Log Renditen:

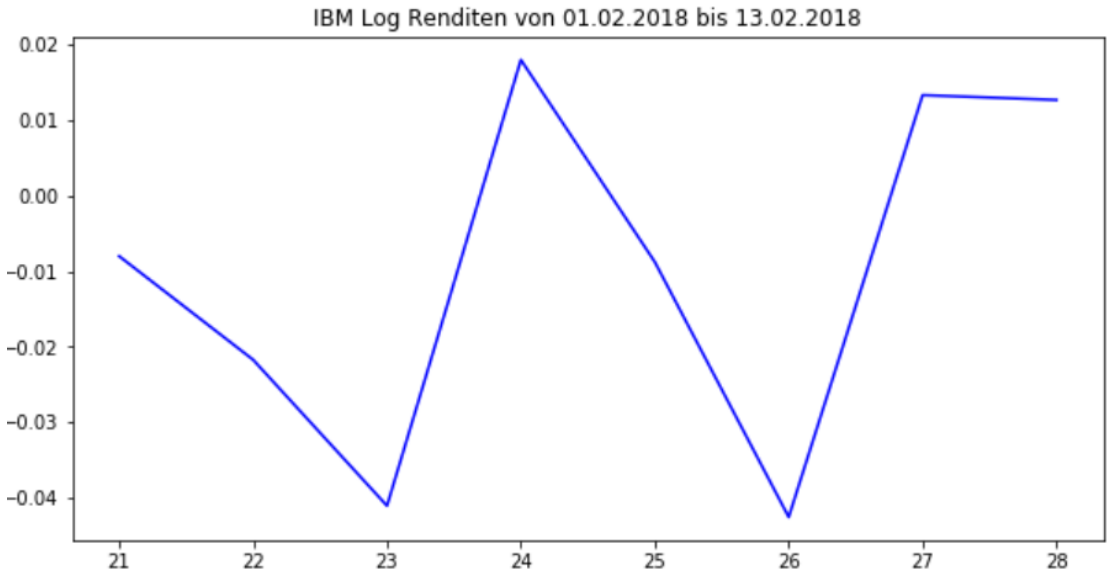

Abbildung 30: Verlauf der IBM Log Renditen von Anfang Februar bis Mitte Februar

Analog zum Abschnitt zu Langzeitprognosen, erhalten wir folgende Tabelle mit den prognostizierten Renditen.

	<OPEN>	<HIGH>	<LOW>	<CLOSE>	H-L	O-C	Log Renditen	Log Renditen(t+1)
21	163.20	164.13	161.94	162.43	2.19	-0.77	-0.008033	-0.021720
22	161.34	161.79	158.87	158.94	2.92	-2.40	-0.021720	-0.041034
23	158.10	158.37	150.00	152.55	8.37	-5.55	-0.041034	0.017931
24	150.52	155.48	149.25	155.31	6.23	4.79	0.017931	-0.008795
25	154.16	155.28	153.43	153.95	1.85	-0.21	-0.008795	-0.042529
26	152.32	153.16	147.50	147.54	5.66	-4.78	-0.042529	0.013264
27	148.84	150.54	144.41	149.51	6.13	0.67	0.013264	0.012628
28	150.81	152.39	150.30	151.41	2.09	0.60	0.012628	-0.004103

Tabelle 8: Python Output der prognostizierten logarithmierten Renditen bei der Kurzzeitprognose

Für die Abschlusskurse vom 01. Februar 2018 gewählt bis zum 13. Februar, welchen man in der nachfolgenden Graphik sieht

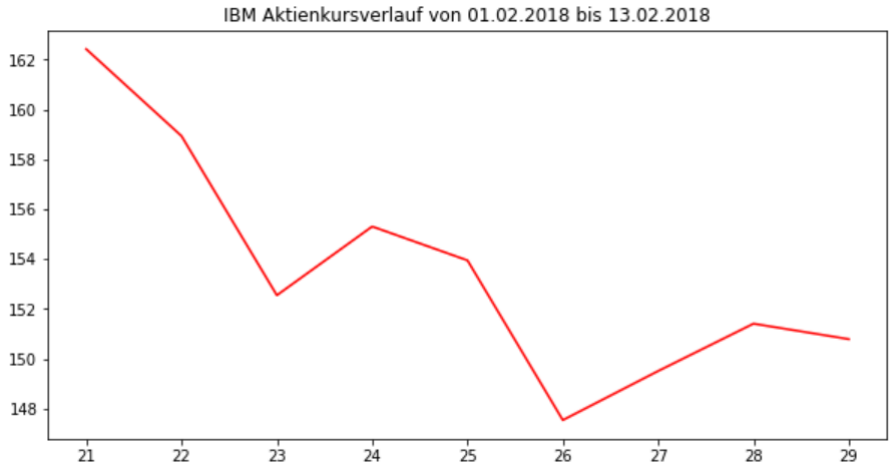

Abbildung 31: Verlauf der IBM Schlusskurse von Anfang Februar bis Mitte Februar

erhalten wir den Output:

	<OPEN>	<HIGH>	<LOW>	<CLOSE>	H-L	O-C	Close(t+1)
21	163.20	164.13	161.94	162.43	2.19	-0.77	158.94
22	161.34	161.79	158.87	158.94	2.92	-2.40	152.55
23	158.10	158.37	150.00	152.55	8.37	-5.55	155.31
24	150.52	155.48	149.25	155.31	6.23	4.79	153.95
25	154.16	155.28	153.43	153.95	1.85	-0.21	147.54
26	152.32	153.16	147.50	147.54	5.66	-4.78	149.51
27	148.84	150.54	144.41	149.51	6.13	0.67	151.41
28	150.81	152.39	150.30	151.41	2.09	0.60	150.79

Tabelle 9: Python Output der prognostizierten Schlusskurse bei der Kurzzeitprognose

Die Parameter der Hidden Layer und der Optimierungsfunktion unseres neuronalen Netzes wurden ausreichend variiert. Dennoch liefern die besten Ergebnisse, das heißt die mit dem kleinsten MSE Werten, dieselben Parameter, welche schon bei den Langzeitprognosen berechnet wurden. Daher wurden diese auch hier bei der Kurzzeitprognose beibehalten. Im Folgenden wollen wir die Prognosen für logarithmierten Renditen und Aktienschlusskurse mit den dazugehörigem Abbildungen wie schon bei den Langzeitprognosen zeigen.

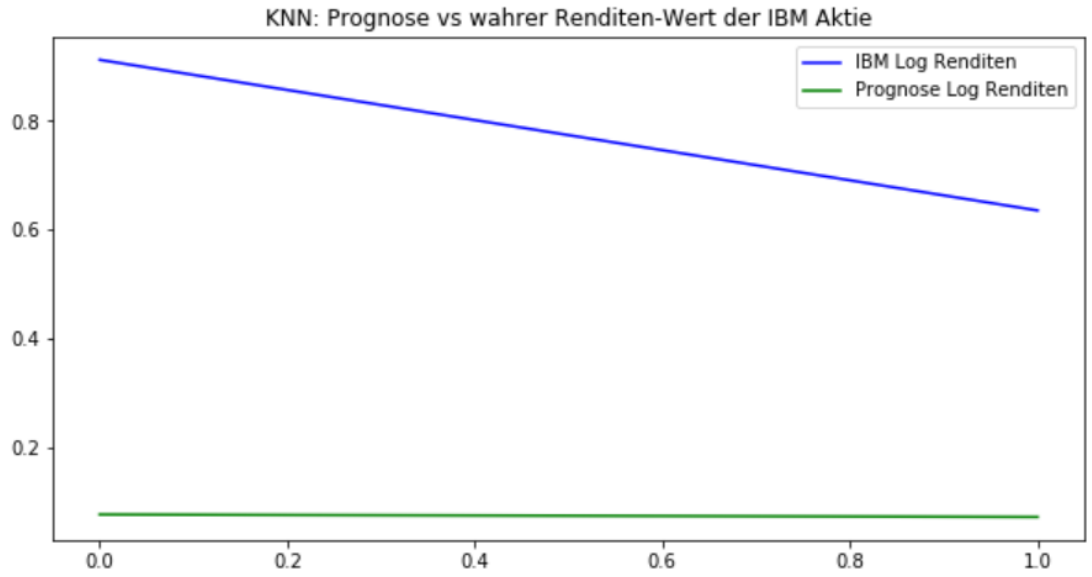

Abbildung 32: Tatsächlicher Verlauf der Renditen und der prognostizierte Verlauf (KNN) bei der Kurzzeitprognose

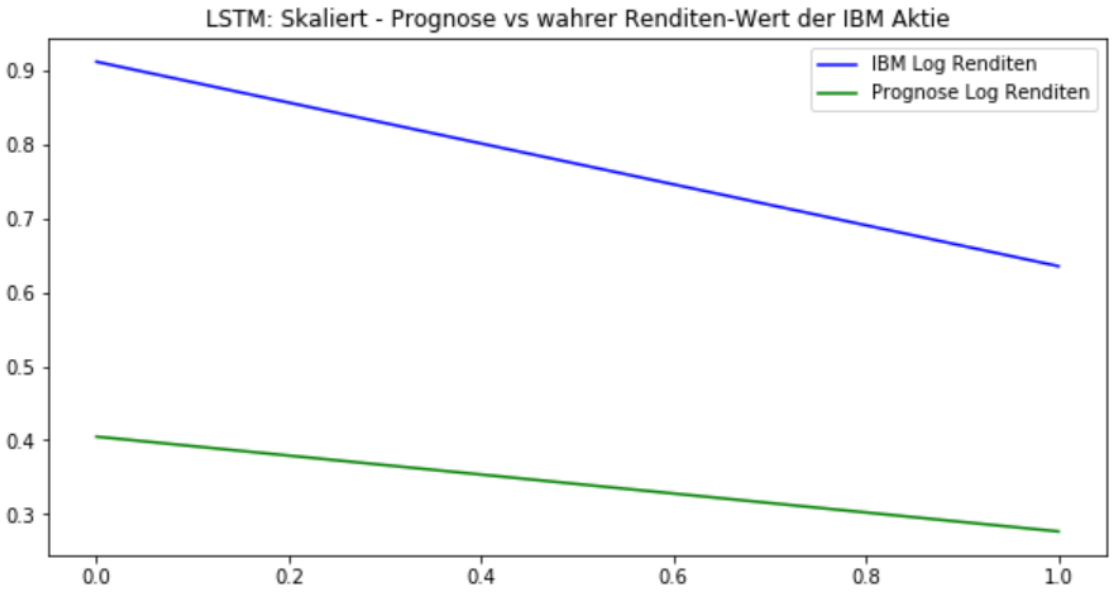

Abbildung 33: Tatsächlicher Verlauf der Renditen und der prognostizierte Verlauf (LSTM) bei der Kurzzeitprognose

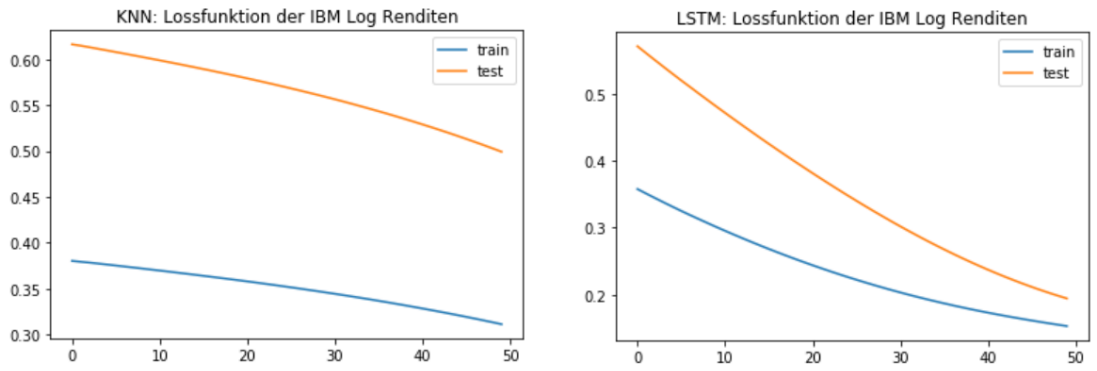

Abbildung 34: Plot der Loss-Funktion für die Kurzzeitprognosen bei den Log Returns: KNN und LSTM

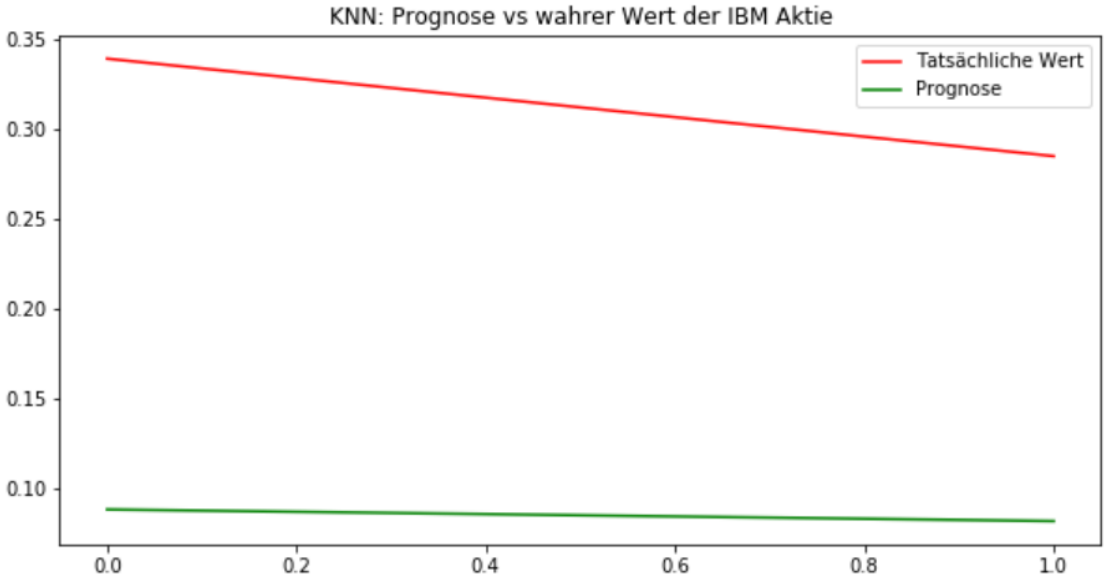

Abbildung 35: Tatsächlicher Verlauf der Schlusskurse und der prognostizierte Verlauf (KNN) bei der Kurzzeitprognose

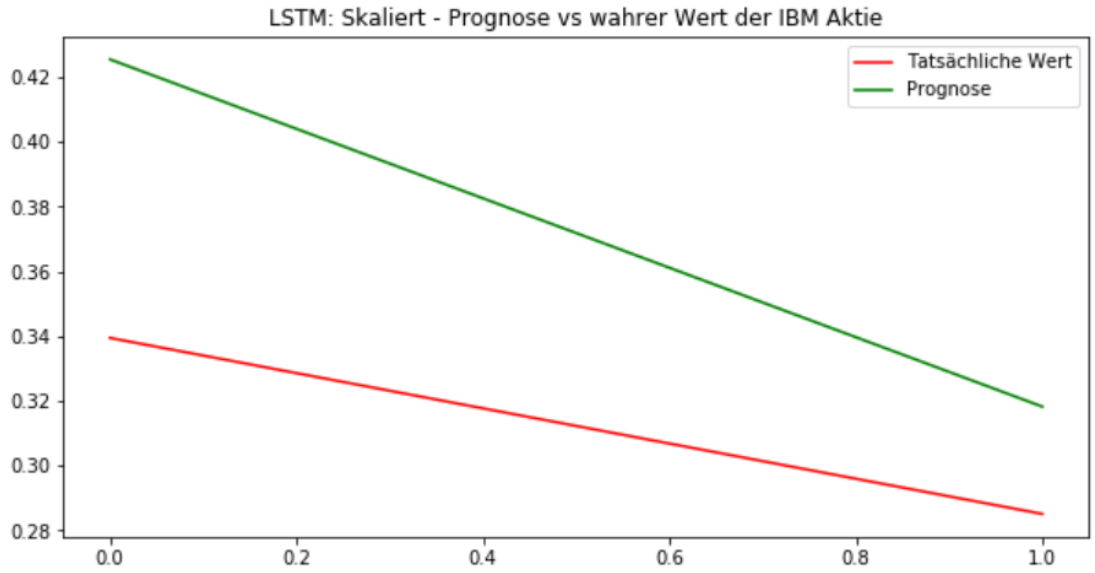

Abbildung 36: Tatsächlicher Verlauf der Schlusskurse und der prognostizierte Verlauf (LSTM) bei der Kurzzeitprognose

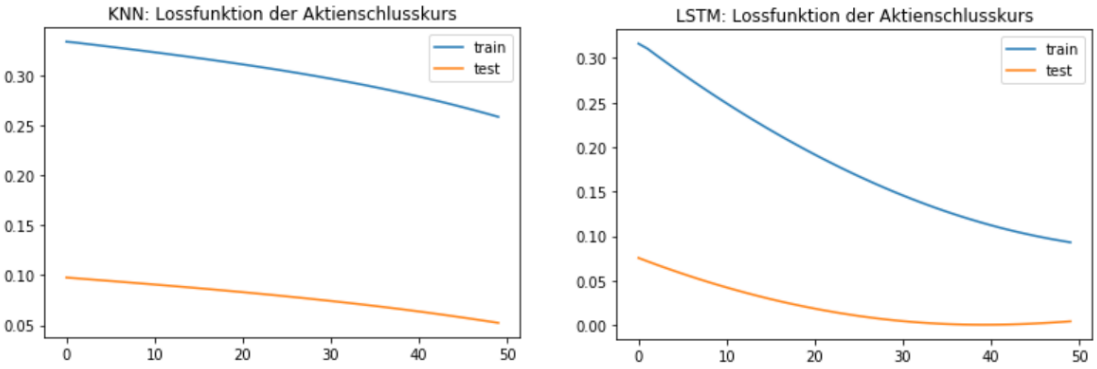

Abbildung 37: Plot der Loss-Funktion für die Kurzzeitprognosen bei den Schlusskursen: KNN und LSTM

Die abgebildeten Resultate zeigen, dass LSTM Netze bei der Kurzzeitprognose deutlich besser abschneiden als lediglich reine Feedforward Netze. Zusätzlich wurden selbstverständlich die MSE Werte verglichen und jedes Mal konnten die LSTM Netze bessere Prognosen liefern als die Feedforward Netze. Die LSTM Netze konnten hier bei der Kurzzeitprognose den Verlauf der logarithmierten Renditen und Aktienschlusskursen, also ob diese steigen oder fallen werden, gut vorhersagen.

2.6 Künstliche neuronale Netze in Mathematica

Der Vollständigkeit halber wollen wir unsere Versuche ein neuronales Netz in *Mathematica* zu implementieren vorstellen. Dabei wollen wir ausgehend von einem Beispiel am Sinus und eines selbst erstellten Moduls zur Prognose dessen, einen Übergang zu unserem Aktienkursprognose-Modell schaffen. Da neuronale Netze in der Dokumentation von *Mathematica* sehr spärlich vertreten sind und wir unser eigenes Programm noch nicht optimalisiert haben, werden wir diesen Abschnitt sehr kurz halten.

2.6.1 Anwendung am Beispiel vom Sinus

Für das Modul `MyNetPrediction`, welches wir programmiert haben, brauchen wir als Eingabewerte unsere Daten und die Länge der Prognose. Wir benutzen das erläuterte RNN Netz.

```
MyNetPrediction[Data_, LengthIn_] :=
Module[{DataTraining, Net, NetTrained, DataNetVerify},
DataTraining =
RandomSample[Map[List /@ Drop[#, -1] -> Take[#, {-1}] &,
Partition[Take[Data, {LengthIn, -1}], LengthIn + 1, 1]]];
Net = NetChain[{GatedRecurrentLayer[10], LinearLayer[1]},
"Input" -> {LengthIn, 1}, "Output" -> 1];
NetTrained = NetTrain[Net, DataTraining];
DataNetVerify =
Flatten@NestList[Append[Drop[#, 1], NetTrained[#]] &,
List /@ Take[Data, {1, LengthIn}], Length[Data] - LengthIn][[
All, -1]];
ListLinePlot[{DataNetVerify,
Take[Data, {LengthIn, Length[Data] - LengthIn}]},
PlotLegends -> {"Vorhersage", "Sin[x]"}]
];
```

Startet man das Modul, so erscheint ein Fenster, welches den Trainingsprozess dynamisch darstellt. Wir können zu jeder Zeit das Training stoppen. In der folgenden Abbildung ist der genannte Trainingsprozess zu sehen.

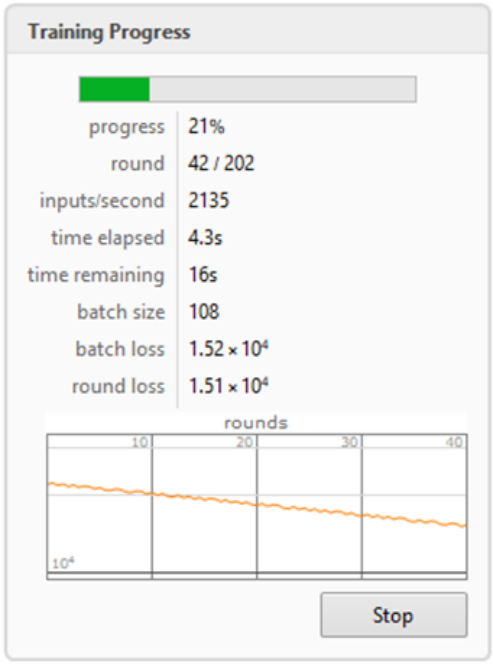

Abbildung 38: Status des Trainings bei neuronalen Netzen in Mathematica

Nun wollen wir die Unterschiede der Prognosen bei verschiedenen Stoppzeiten des Trainings betrachten. Stoppen wir das Training bei 4%, so erhalten wir folgende Prognose.

```
MyNetPrediction[Table[Sin[x], {x, 0, 50, 0.04}], 50]

Out:=
```

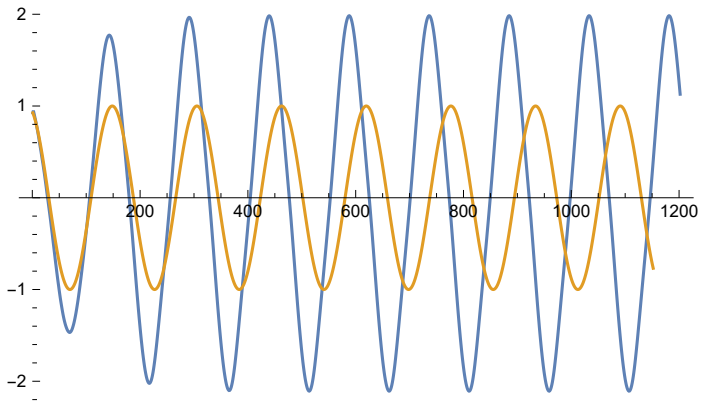

Abbildung 39: Sinus-Prognose (blau) bei einem Training von 4%

Es ist zu erkennen, dass der Verlauf der Prognose dem Sinus sehr ähnelt. Die Schwingungen der Prognose bewegen sich jedoch in einem größeren Intervall.

Im Folgenden sehen wir die Prognose mit einem Stopp des Trainings bei 10 %.

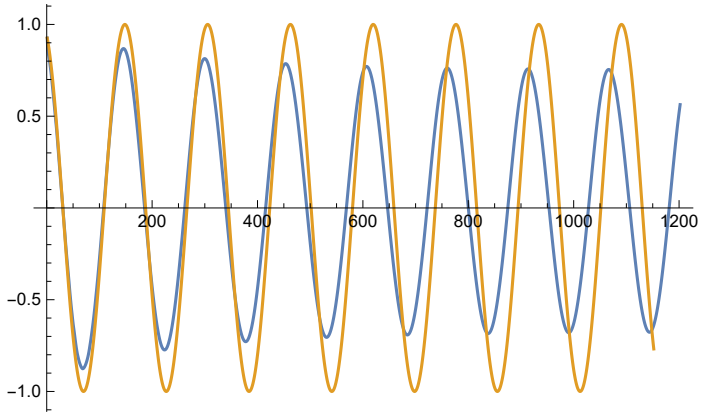

Abbildung 40: Sinus-Prognose (blau) bei einem Training von 10%

Die Prognose ähnelt diesmal mehr dem Sinus und das Intervall in dem sich die Schwingungen bewegen, ist in diesem Fall nun kleiner.

Bei einem Training von 16% erhalten wir folgende Prognose:

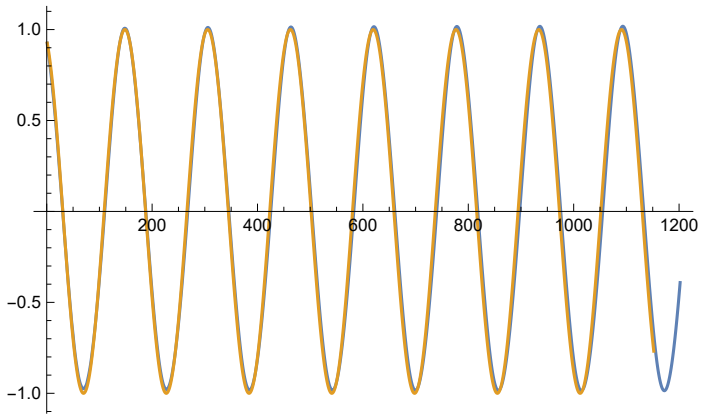

Abbildung 41: Sinus-Prognose (blau) bei einem Training von 16%

Diese Prognose ist so gut wie gleich zum tatsächlichen Verlauf. Jedoch ist dieses Verhalten nicht wünschenswert. Wir replizieren hier den tatsächlichen Verlauf, was nicht der Sinn einer Prognose ist. Dieses Verhalten nennt man auch *overfitting* und soll immer vermieden werden.

2.6.2 Anwendung am Beispiel der Aktienkurse

Unser neuronales Netz haben wir im folgenden Modul `MyNetPrediction2` insofern angepasst, dass wir nun ein Feedforward Netz zur Prognose verwenden.

```
MyNetPrediction2[Data_, LengthIn_] :=
Module[{DataTraining, Net, NetTrained, DataNetVerify},
DataTraining =
RandomSample[Map[List /@ Drop[#, -1] -> Take[#, {-1}] &,
Partition[Take[Data, {LengthIn, -1}], LengthIn + 1, 1]]];
Net = NetChain[{LinearLayer[200], LinearLayer[1]},
"Input" -> {LengthIn, 1}, "Output" -> 1];
NetTrained = NetTrain[Net, DataTraining];
DataNetVerify =
Flatten@NestList[Append[Drop[#, 1], NetTrained[#]] &,
List /@ Take[Data, {1, LengthIn}], Length[Data] - LengthIn][[
All, -1]];
Return[Take[DataNetVerify, -LengthIn]]
];

ListLinePlot[{MyNetPrediction2[Tag17, Length@Tage18], Tage18}]

Out:=
```

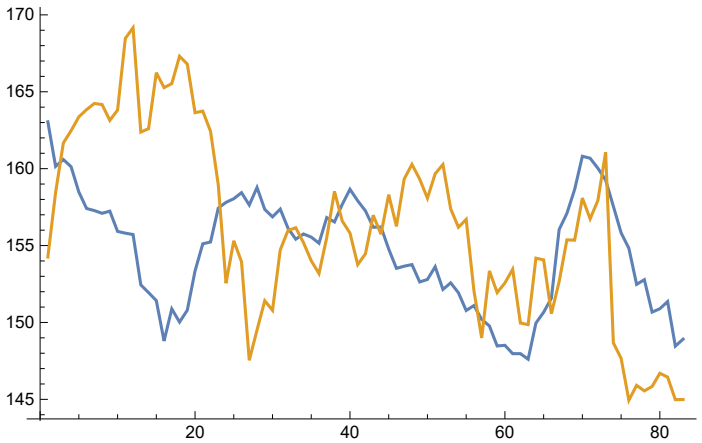

Abbildung 42: Prognose der Aktienkurse (gelb) mittels neuronalen Netzen (blau)

Die Prognose ist schon am Anfang alles andere als optimal. Für den Ergebnisteil werden wir aus diesem Grund lediglich die Prognosen der neuronalen Netze in Python berücksichtigen.

3 Gütemaße zur Bewertung der Modelle

In diesem Abschnitt geht es um diverse Gütemaße, welche wir verwenden, um die Qualität der Modelle zu beurteilen. Damit wir eine objektive Auswertung haben, werden wir folglich fünf Gütemaße ausführen und näher erläutern. Dabei soll A_t immer der echte Aktienkurswert und F_t der prognostizierte Wert zum Zeitpunkt t sein. Unter den Definitionen ist unser *Mathematica* Code bzgl. der Gütemaße aufgeführt. Sämtliche Definitionen der Gütemaße stammen aus dem Artikel von Spyros (vgl. [9])

Definition 3.1 Mean absolute error (MAE).

$$\text{MAE} = \frac{1}{n}\sum_{t=1}^{n}|A_t - F_t|$$

```
MAE[EchtDaten_, Prognose_] :=
Module[{A = EchtDaten, F = Prognose, n = Length[EchtDaten]},
Total[Abs[(A - F)]]/n]
```

Der MAE misst den mittleren absoluten Fehler und ist das elementarste Gütemaß. Ob die Abweichung dabei positiv oder negativ ist, spielt bei diesem Gütemaß keine Rolle. Der Informationsgehalt ist sehr niedrig und das Gütemaß wird oftmals bei heuristischen Auswertungen hinzugenommen. Die Zeitreihe mit dem niedrigsten Fehlerwert ist nach dem MAE die beste Vorhersage.

Definition 3.2 Mean absolute percentage error (MAPE).

$$\text{MAPE} = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{A_t - F_t}{A_t}\right|$$

```
MAPE[EchtDaten_, Prognose_] :=
Module[{A = EchtDaten, F = Prognose, n = Length[EchtDaten]},
Total[Abs[(A - F)/A]]/n]
```

Der MAPE gibt die Güte des Modells in Prozent an und ist eines der populärsten Gütemaße. Die Zeitreihe mit dem niedrigsten Prozent-Fehlerwert ist nach dem MAPE die beste Vorhersage.

Es gilt jedoch zu beachten, dass $A_t = 0 \quad \forall t \in \mathbb{N}$, aufgrund der Division durch Null, nicht definiert ist. Zudem werden negative Fehler (also $A_t < F_t$) stärker gewichtet als positive Fehler (vgl. [9, Seite 527-529]). Diese unsymmetrische Fehlergewichtung tritt in den folgenden drei Gütemaßen nicht auf.

Definition 3.3 Mean arctangent percentage error (MAAPE).

$$\text{MAAPE} = \frac{1}{n}\sum_{t=1}^{n}\arctan\left(\left|\frac{A_t - F_t}{A_t}\right|\right)$$

```
MAAPE[EchtDaten_, Prognose_] :=
Module[{A = EchtDaten, F = Prognose, n = Length[EchtDaten]},
Total[ArcTan[Abs[(A - F)/A]]]/n]
```

Der MAAPE ist eine Abwandlung des MAPE, welcher die schlechten Eigenschaften bereinigt. Ersetzt man die Nullwerte A_t durch einen sehr kleinen numerischen Wert wie 10^{-7}, so geht der Fehlerwert

für den MAPE gegen unendlich, was sehr unpraktisch ist. Dahingegen konvergiert der Fehlerwert bei selbem Sachverhalt für den MAAPE gegen den Wert $\frac{\pi}{2}$.

Da die Fehlerwerte beim MAAPE jedoch im Bogenmaß angegeben werden, ist das Gütemaß nicht so intuitiv wie der MAPE. Die Zeitreihe mit dem niedrigsten Fehlerwert ist nach dem MAAPE die beste Vorhersage

Definition 3.4 Mean absolute scaled error (MASE).

$$\text{MASE} = \frac{\sum_{t=1}^{n} |A_t - F_t|}{\frac{n}{n-1} \sum_{t=2}^{n} |A_t - A_{t-1}|}$$

```
MASE[EchtDaten_, Prognose_] :=
Module[{A = EchtDaten, F = Prognose, n = Length[EchtDaten]},
Total[Abs[A - F]]/((n/(n - 1))*Total[Abs[Differences[A]]])]
```

Der MASE ist ein relativ neues Gütemaß.. Es hat viele hilfreiche Eigenschaften wie beispielsweise die Skaleninvarianz. So kann beim Aufkommen von Nullwerten im Datensatz eine Skalierung durchgeführt werden, welche die Auswertung in keinster Weise beeinträchtigt. Zudem hat es neben der Eigenschaft der symmetrischen Fehlergewichtung noch viele weitere Vorteile, sodass es heutzutage ein sehr wichtiges Gütemaß ist.

Die Zeitreihe mit dem niedrigsten Fehlerwert ist nach dem MASE die beste Vorhersage.

Definition 3.5 Mean directional accuracy (MDA).

$$\text{MDA} = \frac{1}{n} \sum_{t=1}^{n} \mathbb{1}_{\text{sign}(A_t - A_{t-1}) == \text{sign}(F_t - F_{t-1})},$$

wobei die Signumfunktion $\text{sign}(x)$ definiert ist als:

$$\text{sign}(x) := \begin{cases} +1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$$

```
MDA[EchtDaten_, Prognose_] :=
Module[{A = EchtDaten, F = Prognose, n = Length[EchtDaten]},
1/n*Count[Sign[Differences[A]] - Sign[Differences[F]], 0] // N]
```

Der MDA lässt sich nicht aus den zuvor vorgestellten Gütemaßen herleiten. Im Gegensatz zu den anderen Gütemaßen fokussiert sich der MDA auf die Änderungen in einer Zeitreihe, also Abwärts- und Aufwärtsbewegungen. Der MDA gibt die Wahrscheinlichkeit dafür an, dass das unterstellte Modell in einem Zeitschritt die selbe Bewegung hat (auf oder ab) wie die tatsächliche Zeitreihe.

Die Zeitreihe mit dem größten Prozentwert ist nach dem MDA die beste Vorhersage.

4 Ergebnisse

4.1 Vergleich und Bewertung der Modelle M1 bis M4

In diesem Kapitel geht es um die Bewertung, den quantitativen und qualitativen Vergleich der Modelle M1 bis M4. Dazu gehören: Random Walk ohne Drift, Random Walk mit angepasster Verteilungsannahme, Geometrische brownsche Bewegung sowie das ARIMA Modell. Die Auswertungen der neuronalen

Netze werden wir im nächsten Abschnitt genauer betrachten und mit den Ergebnissen aus diesem Abschnitt gegenüberstellen.

Mit den soeben kennengelernten Gütemaßen können wir nun qualitative Aussagen über die Modelle machen. Als Prognose jenes Modelles, nehmen wir den durch die Monte-Carlo-Simulation (vgl. [2, Seite 2]) gewonnenen Pfad. Dabei simulieren wir insgesamt 100.000 Durchläufe und nehmen den Mittelwert zu jedem Zeitpunkt. Dies ist für alle Modelle, außer dem ARIMA Modell, notwendig, da dieses nicht durch eine Simulation entstanden ist.

```
progM1 = Mean[Table[Modell1[Last[KursTage17], Length[KursTage18], sigmaTage17, muTage17], 100000]];

progM2 = Mean[Table[Modell2[Last[KursTage17], Length[KursTage18], sigmaTage17, muTage17], 100000]];

progM3 = Mean[Table[Modell3[Last[KursTage17], Length[KursTage18], sigmaTage17, muTage17], 100000]];

progM4 = ARIMA[KursTage17, Length[KursTage18]];
```

In folgender Abbildung sind alle Prognosen und der tatsächliche Werteverlauf zu sehen.

```
Show[
ListLinePlot[progM1, PlotStyle -> Orange],
ListLinePlot[progM2, PlotStyle -> Green],
ListLinePlot[progM3, PlotStyle -> Red],
ListLinePlot[progM4, PlotStyle -> Black],
ListLinePlot[KursTage18],
PlotRange -> All]
```

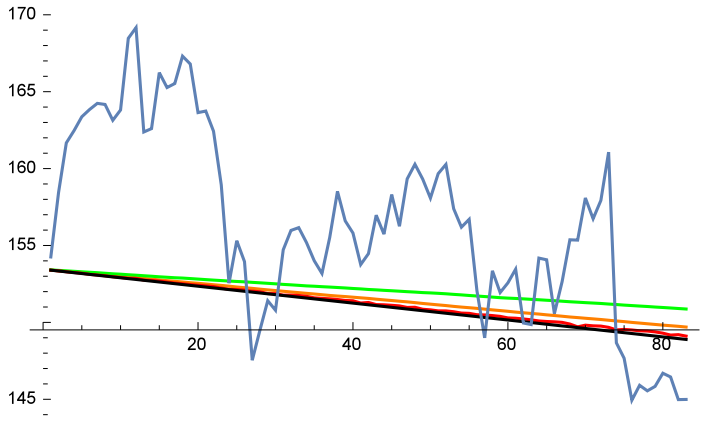

Abbildung 43: tatsächlicher Verlauf (blau), M1 (orange), M2 (grün), M3 (rot), M4 (schwarz)

Für die Auswertung haben wir noch ein weiteres *Mathematica*-Modul implementiert, welches die Güte jedes Modells unter den verschiedenen Gütemaßen auswertet. Dabei wird in der letzten Spalte zusätzlich das beste Modell unter den jeweiligen Gütemaß ausgegeben. Im Folgenden ist der Code unseres Moduls `GüteStatistik[]` zu entnehmen:

```
GueteStatistik[m1_, m2_, m3_, m4_] := Module[{},

mape = {"MAPE", MAPE[m1,KursTage18], MAPE[m2,KursTage18], MAPE[m3,KursTage18], MAPE[m4,KursTage18]};
mape = Append[mape, {"Modell ", Flatten[Position[mape, Min[Drop[mape, 1]]] - 1]}];

mda = {"MDA", MDA[m1, KursTage18], MDA[m2, KursTage18], MDA[m3, KursTage18], MDA[m4, KursTage18]};
mda = Append[mda, {"Modell ", Flatten[Position[mda, Max[Drop[mda, 1]]] - 1]}];

mae = {"MAE", MAE[m1, KursTage18], MAE[m2, KursTage18], MAE[m3, KursTage18], MAE[m4, KursTage18]};
mae = Append[mae, {"Modell ", Flatten[Position[mae, Min[Drop[mae, 1]]] - 1]}];

maape ={"MAAPE",MAAPE[m1,KursTage18],MAAPE[m2,KursTage18],MAAPE[m3,KursTage18],MAAPE[m4,KursTage18]};
maape = Append[maape, {"Modell ", Flatten[Position[maape, Min[Drop[maape, 1]]] - 1]}];

mase = {"MASE", MASE[m1,KursTage18], MASE[m2,KursTage18], MASE[m3,KursTage18], MASE[m4,KursTage18]};
mase = Append[mase, {"Modell ", Flatten[Position[mase, Min[Drop[mase, 1]]] - 1]}];
```

```
Guete = MatrixForm@{{"␣", "Modell␣1", "Modell␣2", "Modell␣3",
"Modell␣4", "Bestes␣Modell"}, mape, mda, mae, maape, mase}
]
```

Als Eingabewerte benutzen wir die durch Monte-Carlo-Simulation gewonnenen Pfade bzw. Prognosen, also:

```
GueteStatistik[progM1, progM2, progM3, progM4]
```

Als Ergebnis erhalten wir folgende Matrix:

$$\begin{pmatrix} & \text{Modell 1} & \text{Modell 2} & \text{Modell 3} & \text{Modell 4} & \text{Bestes Modell} \\ \text{MAPE} & 0.0391 & 0.0376 & 0.0398 & 0.0401 & \{\text{Modell}, \{2\}\} \\ \text{MDA} & 0.4698 & 0.4698 & 0.5060 & 0.4698 & \{\text{Modell}, \{3\}\} \\ \text{MAE} & 5.9393 & 5.7351 & 6.0477 & 6.0901 & \{\text{Modell}, \{2\}\} \\ \text{MAAPE} & 0.0390 & 0.0375 & 0.0398 & 0.0401 & \{\text{Modell}, \{2\}\} \\ \text{MASE} & 131.43 & 184.91 & 104.83 & 110.89 & \{\text{Modell}, \{3\}\} \end{pmatrix}$$

Es ist zu erkennen, dass Modell 2, also der Random Walk mit angepasster Verteilungsannahme, unter den Gütemaßen MAPE, MAE, MAAPE das beste Modell ist. Das diese drei Gütemaße das selbe Modell vorschlagen ist nicht verwunderlich. Der MAPE ist eine Erweiterung des MAE und der MAAPE ist wiederum eine Abwandlung des MAPE, sodass diese Gütemaße im weitesten Sinne miteinander verwandt sind.

Unter dem Gütemaß MASE erhalten wir das Modell 3, also die Geometrische brownsche Bewegung, als das beste Modell. Der MDA liefert auch das Modell 3. Jedoch ist die Aussage des MDA in diesem Fall nicht sehr relevant. In der Abbildung 43 ist zu erkennen, dass die Prognosen, aufgrund der Monte-Carlo-Simulation, nahezu linear verlaufen. In fast allen Zeitschritten findet eine Abwärtsbewegung statt, sodass es kaum Dynamik gibt. Der MDA bewertet jedoch diese Dynamik und deswegen sind diese Ergebnisse außer Acht zu lassen. Im folgenden Ergebnisteil wollen wir nun die soeben verifizierten Ergebnisse mit denen der neuronalen Netze vergleichen.

4.2 Bewertung der Prognose von neuronalen Netzen

In diesem Abschnitt wollen wir vergleichen, ob das Modell und die Ergebnisse der neuronalen Netze aus Abschnitt 2.5.2 besser sind als die Modelle M1 bis M4. Wir importieren die in Python berechnete Prognose in *Mathematica*, um unsere programmierten Gütemaße zu nutzen.

Berechnen wir nun die Güte für das LSTM Netz unserer Langzeitprognose, so erhalten wir folgendes Ergebnis:

Gütemaß	Wert
MAPE	0.02863
MDA	0.45679
MAE	4.55181
MAAPE	0.02860
MASE	2.02416

Tabelle 10: Gütestatistik zu LSTM Netzen

Vergleicht man die Tabelle mit der Ergebnismatrix aus dem vorigen Abschnitt 4.1, so ist zu erkennen, dass die LSTM Netze aus Python - begründet durch die Gütemaße MAPE, MAE, MAAPE und MASE - ein viel besseres Ergebnis liefern als die Modelle M1 bis M4. Schaut man sich beispielsweise den MAPE genauer an, so liegt dieser bei den Modellen M1 bis M4 um den Wert 0.4. Das heißt, die Prognose weicht im Durchschnitt um 40% ab. Bei den neuronalen Netzen liegen wir bei einem Durchschnittswert von 28%, was qualitativ eine enorme Verbesserung ist.

Bezüglich dem MDA ist das LSTM Netz schlechter als alle anderen Modelle. Wie soeben schon erwähnt, ist der MDA jedoch zu vernachlässigen. Unser Aktienkursverlauf im Jahr 2018 fällt tendenziell ab. Auf diesen Sachverhalt basieren die Modelle M1 bis M4, sodass wir dort eine nahezu lineare Prognose kriegen, welche fällt. Das LSTM Netz hat eine stärkere Dynamik und wird dafür im MDA „bestraft“ Zusätzlich ist die Aussagekraft des MDA nicht so stark wie die der anderen Gütemaße. Beim MDA geht es lediglich um die Steigung zwischen zwei Zeitpunkten. Die Höhe der Steigung spielt dabei keine Rolle.

Die anderen vier Gütemaße berechnen quantitativ den Vorhersage-Fehler und haben somit eine viel größere Aussagekraft zur Bewertung der Qualität der Modelle. Die LSTM Netze sind nach diesen vier Gütemaßen den anderen Modellen deutlich überlegen.

5 Fazit und Ausblick

Die Durchführung des Projektes hat uns ermöglicht die theoretischen Modelle für Prognosen, welche wir im Laufe des Master-Studiums kennengelernt haben, praktisch umzusetzen. Dabei haben wir die Erfahrung gemacht, dass eine sehr gute Langzeitprognose kaum umzusetzen ist. Beispielsweise führen die Modelle M1 bis M4 lediglich den durchschnittlichen Trend aus der Vergangenheit wieder und das bezeichnen wir dann als Prognose.

Alle Prognosen unterliegen einem Modell. Modelle hingegen sind durch sämtliche Annahmen aufgebaut, welche meistens auch nicht stimmen. Ein Extrembeispiel ist beispielsweise das Modell M1, also der Random Walk ohne Drift. Bei diesem Modell liegt die Annahme normalverteilter Renditen zu Grunde. Dass diese Annahme nicht stimmt, konnten wir in dieser Arbeit widerlegen. Letztendlich ist der Aktienmarkt viel zu komplex, um ein vereinfachtes Modell zu konstruieren. Es gibt zu viele Parameter, die man nicht auf ein einfaches Modell runterbrechen kann. Neben dem Angebot und der Nachfrage bestimmen auch die Zinsentwicklungen, das Wechselkursniveau oder auch die Medien anteilig den Verlauf eines Aktienkurses. Daher kommen wir zu dem Entschluss, dass es in naher Zukunft kein einfaches Modell geben wird, welches Aktienkurse vorhersagt.

Mithilfe der verschiedenen Gütemaße konnten wir desto trotz die Modelle qualitativ bewerten. Es hat sich dabei herausgestellt, dass die neuronalen Netze den anderen Modellen sehr überlegen sind. Zudem hat uns die Einarbeitung in den neuronalen Netze eine Menge Spaß bereitet, da diese sehr vielfältig sind. Auch gab es in den letzten Jahren einen sehr großen Fortschritt im Bereich der neuronalen Netze, sodass wir uns vorstellen können, dass diese im Bereich der Kurzzeitprognosen einen großen Einfluss haben werden.

Da wir uns in dieser Arbeit auf die Langzeitprognosen spezialisiert haben, könnte man im nächsten Schritt den Beobachtungszeitraum verringern und Prognosen im Sekundenbereich durchführen. Wir können uns sehr gut vorstellen, dass Prognosen im Sekundenbereich wesentlich besser sind, da wir damit eine viel höhere Informationsdichte besitzen. Betrachtet man nur die Tagesschlusskurse, so geht die restliche Information des ganzen Tages verloren. Als Ausblick auf weitere Arbeiten würden wir aus diesem Grund vorschlagen, die aktuellsten neuronalen Netze auf Prognosen im Sekundenbereich zu analysieren.

Literaturverzeichnis

Verzeichnis der selbstständigen Literaturquellen

[1] FRIEDRICH, A. *Neuronale Netze. Theoretische Grundlagen und Anwendung in der Verkehrszeichenerkennung.* München: GRIN Verlag GmbH, 2011. ISBN 978-3-6408-6246-7.

[2] GLASSERMAN, P. *Monte Carlo methods in financial engineering*. New York: Springer, 2004. Applications of mathematics. 53. ISBN 978-0-3870-0451-8.

[3] HULL, J. *Options, futures and other derivatives*. 7., int. ed. Upper Saddle River,NJ: Pearson Education, 2009. Wirtschaft. ISBN 978-0-1350-0994-9.

[4] SHUMWAY, R.H. und D.S. STOFFER. *Time Series Analysis and Its Applications. With R Examples*. 4th ed. 2017. Cham: Springer International Publishing; Springer International Publishing AG, 2017. Springer Texts in Statistics. ISBN 978-3-3195-2451-1.

Verzeichnis der unselbstständigen Literaturquellen

[5] BALZER, P. [Tutorial] Neuronale Netze einfach erklärt [online], 2016 [Zugriff am: 23. August 2018]. Verfügbar unter: http://www.cbcity.de/tutorial-neuronale-netze-einfach-erklaert

[6] BREWER, K.D., Y. FENG und C.C.Y. KWAN. *Geometric Brownian Motion, Option Pricing, and Simulation: Some Spreadsheet-Based Exercises in Financial Modeling* [online], 2012 [Zugriff am: 20. Juni 2018]. Verfügbar unter: https://epublications.bond.edu.au/cgi/viewcontent.cgi?referer=https://www.google.de/&httpsredir=1&article=1131&context=ejsie

[7] DUCASSE, S.G. *Akaike's Information Criterion: Definition, Formulas* [online], 2018 [Zugriff am: 23. August 2018]. Verfügbar unter: http://www.statisticshowto.com/akaikes-information-criterion/

[8] MOESER, J. Künstliche neuronale Netze - Aufbau & Funktionsweise [online], 2017 [Zugriff am: 23. August 2018]. Verfügbar unter: https://jaai.de/kuenstliche-neuronale-netze-aufbau-funktion-291/

[9] SPYROS, M. *Accuracy measures: theoretical and practical concerns* [online]. International Journal of Forecasting, 1993, 9 (4), 527-529. ISSN 0169-2070. Verfügbar unter: https://doi.org/10.1016/0169-2070(93)90079-3

Anhang

A Einlesen der Daten in *Mathematica*

Das Einlesen der Daten kann in `Mathematica` sehr unkompliziert mithilfe des Befehls `Import[]` durchgeführt werden. Der Befehl ist sehr allgemein und ist mit den meisten Dateiformaten kompatibel. Neben *.txt* und *.csv* Dateien, lassen sich auch Bildformate wie *.gif* und *.png* importieren. Der Import erfolgt über die Eingabe des Dateipfades.

```
KursSek17 =
Flatten[Take[Drop[Import[
"C:\\Users\\admin\\Desktop\\DATEN\\IBM\\2017_Komplett\\US1.IBM_170101_171231_Sekunden.csv"],
1], All, -1]];

KursMin17 =
Flatten[Take[Drop[Import[
"C:\\Users\\admin\\Desktop\\DATEN\\IBM\\2017_Komplett\\US1.IBM_170101_171231_Minuten.csv"],
1], All, -1]];

KursStunden17 =
Flatten[Take[Drop[Import[
"C:\\Users\\admin\\Desktop\\DATEN\\IBM\\2017_Komplett\\US1.IBM_170101_171231_Stunden.csv"],
1], All, -1]];

KursTage17 =
Flatten[Take[Drop[Import[
"C:\\Users\\admin\\Desktop\\DATEN\\IBM\\2017_Komplett\\US1.IBM_170101_171231_Tage.csv"],
1], All, -1]];
```

Der Befehl `Import[]` liest unsere *.csv* Datei und speichert diese als Matrix in Mathematica. Da neben den Kursen noch andere Informationen im Datensatz enthalten sind (wie z.B.: Datum oder Uhrzeit), möchten wir diese filtern. Dabei benutzen wir zuerst den Befehl `Drop[]`, welcher die Spaltenbezeichnungen löscht. Danach ziehen wir uns mit dem Befehl `Take[]` die letzte Spalte raus, welche die gewünschten Kurse enthält. Mit dem Befehl `Flatten[]` werden überflüssige Array-Klammern beseitigt, sodass wir am Ende nur noch eine Liste mit unseren Kursen haben.

Analog haben wir die Kurse von Januar 2018 bis April 2018 eingelesen und auf gleicher Weise definiert
(KursSek18, KursMin18, KursStunden18, KursTage18).