Technische Hochschule Rosenheim

Vorhersage von Zeitreihen mithilfe von neuronalen Netzen

John Lyons
Angewendete künstliche Intelligenz
Lehrstuhl für Informatik

SS 22

Abstract

Diese Arbeit richtet sich an Informatiker, die an der Zeitreihenvorhersage mit neuronalen Netzen interessiert sind. Da es heutzutage immer mehr Anwendungsfälle gibt, in denen die künstliche Intelligenz zur Vorhersage von Ereignissen oder konkreten Werten eingesetzt werden kann, werden auch viele Wettbewerbe in diesem Bereich veranstaltet.
So wird anhand einer konkreten Problemstellung eines Wettbewerbs die Datenaufbereitung und deren Probleme aufgezeigt. Dabei stellt sich heraus, dass es viele Möglichkeiten gibt die Daten vorzuverarbeiten und sich das Verfahren auf die Qualität der Vorhersage auswirkt.
Anschließend werden die wichtigsten Grundlagen der neuronalen Netze, wie z.B. das sog. Multilayer Perceptron (MLP), der Backpropagation Algorithmus und Tensoren erklärt. Zudem werden weitere Forschungsergebnisse der letzten Jahre, vor allem im Bereich der Optimierer und Aktivierungsfunktionen, vorgestellt.
Zuletzt werden mögliche neuronale Netzwerkschichten, wie z.B. sog. Long Short-Term Memory (LSTM) oder Convolution Schichten, eingeführt und anhand einer vereinfachten Version der Aufgabenstellung zur Probe gestellt. Dabei stellt sich heraus, dass die LSTM Schicht am besten abschneidet.
Um das perfekte Modell für die Aufgabenstellung des Wettbewerbs zu finden, wird eine Gittersuche ausgeführt, in der eine definierte Anzahl an Parameterkombinationen und Modellen getestet wird. Hierbei schneidet das klassische MLP in Kombination einer Integrierung von Zukunftsdaten am besten ab.
Insgesamt wird aus den Ergebnissen geschlossen, dass der Einsatz der Architektur stark von dem gegebenen Problem und der Menge der Daten abhängt. Ein optimales Ergebnis mit entsprechenden Hyperparametern kann in dieser Arbeit mithilfe einer Gittersuche gefunden werden.

Inhaltsverzeichnis

Acronyms

RNN Recurrent Neural Network

LSTM Long Short-Term Memory

GRU Gated Recurrent Unit

MLP Multilayer Perceptron

ARIMA Auto-Regressive Integrated Moving Average

SARIMA Seasonal Auto-Regressive Integrated Moving Average

NNAR Neural Network Auto-Regressive

XGB Extreme Gradient Boosting

MAE Mean Absolute Error

MSE Mean Squared Error

1 Übersicht

1.1 Einführung

Die Vorhersage von zukünftigen Ereignissen wird heutzutage immer wichtiger. Denn durch eine präzise Schätzung können gezielt Mittel im Voraus geplant und dadurch Kosten gesenkt werden. So können Energiekonzerne die Daten der Vergangenheit nutzen, um mathematische Modelle oder künstliche Intelligenz für die Vorhersagefunktion zu trainieren. Eine Ersparnis liegt dann vor, wenn die gekaufte Menge der Energie oder der jeweiligen Güter so nah wie möglich an dem tatsächlichen Verbrauch liegt. Denn ein Überschuss muss ansonsten anderweitig gelagert oder günstiger weiterverkauft werden.
Für eine lange Zeit wurden mathematische Modelle, wie z.B. Auto-Regressive Integrated Moving Average (ARIMA), für die Vorhersage verwendet. Durch die Zunahme von verfügbaren Daten, Steigung der Hardwareperformanz und Einsatz von Grafikkarten in den letzten zehn Jahren können jetzt neuronale Modelle schneller und mit mehr Daten trainiert werden. Neuste Publikationen präsentieren sowohl Architekturvorschläge als auch konkrete Neuerungen innerhalb der neuronalen Netze, wie z.B. verbesserte sog. Aktivierungsfunktionen. Zusammen mit den bereits lange zur Verfügung stehenden Verfahren bzw. neuronalen Schichten, wie z.B. LSTM Schichten, den Attention Mechanismus, Convolution Schichten und die Transformer Architektur, ist eine Evaluation auf ein spezielles Zeitreihenproblem sinnvoll.
Es werden zunächst die lang bekannten mathematischen Verfahren für Zeitreihenvorhersagen kompakt erklärt und deren Vor- und Nachteile erwähnt. Außerdem wird der zugrundeliegende Datensatz des Wettbewerbs gezeigt und potenzielle Ansätze der Datenaufbereitung diskutiert. Danach werden die wichtigsten neuronalen Architekturen vorgestellt und deren Funktionsweise in Zeitreihen erklärt. Daraufhin werden die gängigen neuronalen Schichten auf ein reduziertes Problem angewendet und ausgewertet. Zum Schluss wird ein Modell für die Vorhersage mithilfe einer Gittersuche optimiert und auf das originale Problem angewendet.
Im nächsten Abschnitt soll das Ziel der Arbeit konkretisiert werden.

1.2 Zielsetzung

In letzter Zeit gibt es viele Publikationen, welche neue Architekturen für eine Zeitreihenvorhersage vorstellen. Diese bestehen aus den bereits genannten Schichten der neuronalen Netze und unterscheiden sich stark voneinander. Beispielsweise variiert sowohl die Wahl der Reihenfolge, die Kombination der Schichten als auch die Datenaufbereitung und viele weitere Hyperparameter, wie z.B. die Anzahl der Neuronen innerhalb einer Schicht.
So soll in der Arbeit von den grundlegenden Bausteinen und Schichten der neuronalen Netze eine schrittweise Annäherung an die komplexeren Architekturen getätigt werden. Dabei wird überprüft, ob die Auswechslung von simpleren Bausteinen durch andere Bausteine automatisch zu einer Verbesserung des Ergebnisses führt.
Zudem soll auf Basis der vorherigen Ergebnisse geprüft werden, ob eine gewählte Referenzarchitektur besser abschneidet als die einfachen Implementierungen. Nach der Evaluierung der einzelnen Lösungen soll das beste Modell gewählt und mithilfe einer sog. Gittersuche optimiert werden.
Als Datensatz wird hierbei der Gasdatensatz von „Trading Hub Europe“ verwendet, welcher im nächsten Abschnitt weiter erklärt wird.

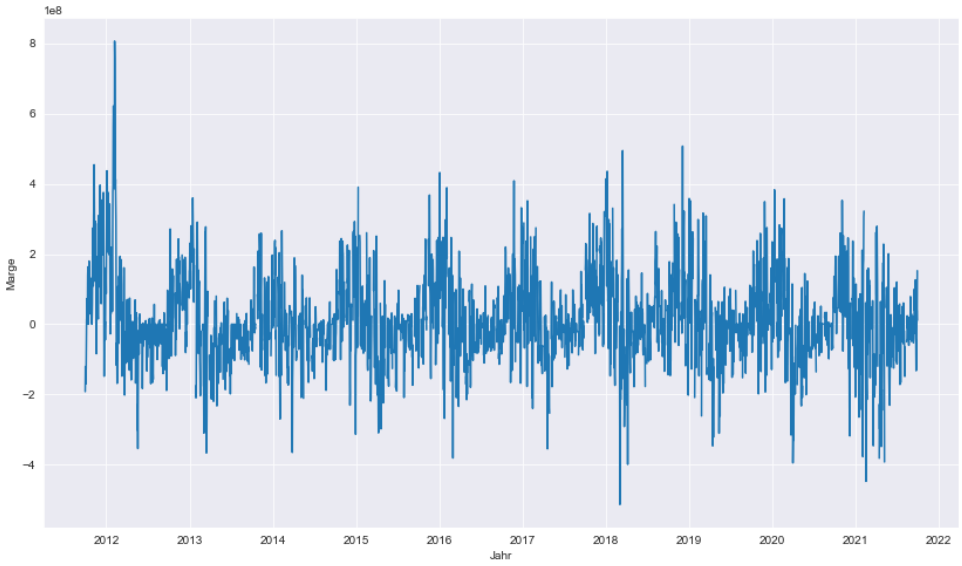


Abbildung 1: Zeitreihe

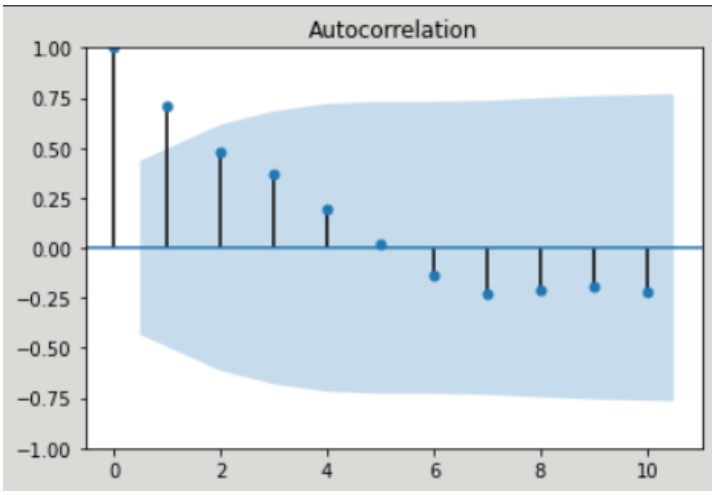
1.3 Zugrundeliegende Zeitreihe des Datensatzes

Abb. 1 zeigt den Verlauf einer Zeitreihe. Die zugrundeliegenden Daten, welche auch im Verlauf dieser Arbeit für die Bewertung der verschiedenen Modelle verwendet werden, stammen von „Trading Hub Europe“. Es ist deutlich zu sehen, dass eine gewissen Periodizität im Datensatz herrscht. So scheint es, als gäbe es eine jährliche Saisonalität und keinen kontinuierlichen Trend. Bei genauerer Analyse ist sogar eine wöchentliche Saisonalität zu erkennen. Dies zeigt die Autokorrelationsfunktion und die Analyse mithilfe der „statsmodels“ Bibliothek in Abb. 2.

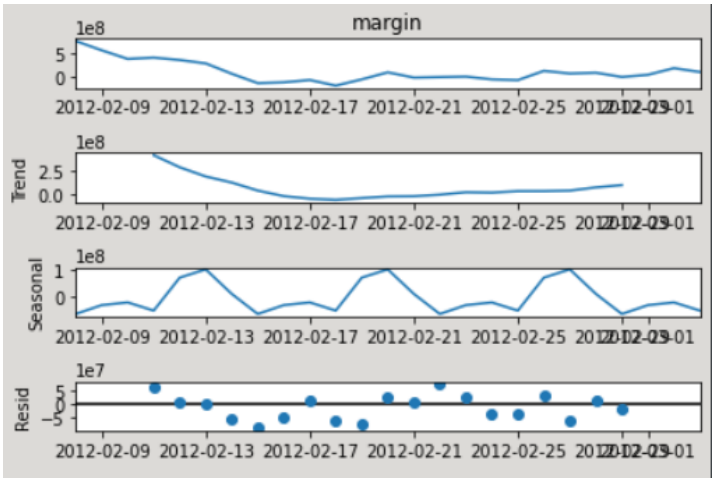
Dabei geht es bei dem Problem implizit um die Vorhersage vom Gasverbrauch in der Industrie und den Haushalten. Die Hauptaufgabe von Trading Hub Europe ist die Balancierung des Gashaushalts im Gasnetz. Überschüssiges Gas muss aufgrund der physischen Kapazität des Netzwerks ausgespeist und fehlendes Gas eingespeist werden.
Die Daten werden in einem 24-Stunden-Intervall bereitgestellt und die wichtigsten Attribute des Datensatzes sind:

- margin
- SystemBuy
- SystemSell
- NK-Saldo
- BK-Saldo

Dabei setzt sich *margin* direkt aus *SystemBuy* und *SystemSell* zusammen. *SystemBuy* ist die Menge an Gas, die am Ende des Tages insgesamt gekauft wurde und *SystemSell* die Menge an Gas, die am Ende des Tages verkauft wurde. Beide Attribute können zu einem Zeitpunkt (d.h. am Ende eines Tages) ungleich Null sein. Dies ist der Fall, da die Balancierung teilweise mehrmals am Tag durchgeführt werden muss, und es deshalb vorkommen kann, dass an einem



(a) Autokorrelationsfunktion



(b) Trend und Saisonalität in der Zeitreihe

Abbildung 2: Analyse der Periodizität mithilfe der statsmodels Bibliothek [sta22]

Tag sowohl gekauft als auch verkauft wurde.
Wenn im Voraus bekannt wäre, wie viel Gas letztendlich am Ende des Tages gehandelt werden müsste, so wäre es möglich an den vorherigen Tagen geschickt zu den günstigsten Zeitpunkten einzukaufen und somit Geld zu sparen.
Die Schwierigkeit der Vorhersage dieser Daten kann auch ohne explorative Datenanalyse begründet werden. Denn die Vorhersage des *margin* Attributs basiert auf anderen exogenen Attributen, wie z.B. *NK-Saldo* und *BK-Saldo* und diese Attribute basieren bereits auf externen Vorhersagen, auf welche kein Einfluss mehr genommen werden kann. So ist *BK-Saldo* die Differenz zwischen dem tatsächlichen Verbrauch von großen Haushalten (Industrie etc.) und der externen Vorhersage. Deshalb muss das Modell (falls es die exogenen Variablen berücksichtigt) implizit eine Vorhersage für eine Schätzung leisten, was sich als schwierig erweist.

Aufgabenstellung des Wettbewerbs

Die beschriebenen Daten werden von Trading Hub Europe im Rahmen des „AICup“ Wettbewerbs der Universität Passau bereitgestellt. Ziel ist es, die o.g. *margin* vorherzusagen. Die

Daten reichen vom 2011-10-01 bis zum 2021-09-30 und haben eine tägliche Frequenz.
Des Weiteren sind zwischen dem 2012-10-24 und dem letzten Datenpunkt Lücken. Diese dienen der Evaluation und sind während des ganzen Wettbewerbszeitraums nicht für die Entwicklung verfügbar. Die Lücken folgen nach jeweils 24 voll angegebenen Tagen und sind sechs Tage lang. Dabei soll jedoch nur der zweite bis inkl. sechste Tag vorhergesagt werden, was die Vorhersage erschwert, da ein Tag zwischen den Vorhersagen liegt.
Bei der Berechnung des Scores werden die Fehler des ersten zu vorhersagenden Tages mit 50 Prozent und die der restlichen vier Tage mit jeweils 0.125 Prozent gewichtet. Inwiefern sich das auf die Wahl des Modells oder der Verlustfunktion auswirkt, wird beispielsweise in Kap. 2.4 diskutiert.
Zudem dürfen nur alle Tage vor dem Vorhersagezeitraum zum Training und zur Inferenz (Vorhersage) genutzt werden, da die zukünftigen Daten Aufschluss über die Lücken geben könnten. Deshalb dient dieses Szenario als perfekte Simulation eines echten Use Cases, wo ebenfalls Daten in der Vergangenheit fehlen und trotz Lücken ein Modell entwickelt werden muss, welches trotz der Bedingungen möglichst zuverlässig in die Zukunft vorhersagen kann.
Die Handhabung der Besonderheiten bezüglich Lücken im Trainingsdatensatz, Übersprung eines Tages bei der Vorhersage und die Gewichtung der Ergebnisse wird in den jeweiligen Kapiteln, vor allem in Kap. 1.5 und Kap. 2.4 diskutiert.

1.4 Mathematische Verfahren zur Vorhersage

Da einige Architekturen der neuronalen Netze den klassischen mathematischen Verfahren in einigen Aspekten ähneln, wird hier kurz auf die bekanntesten stochastischen Verfahren zur Zeitreihenvorhersage eingegangen und ein erster Unterschied aufgezeigt.
Abb. 1 von Kap. zeigt den Verlauf des zu vorhersagenden *margin* Wertes. Mathematisch betrachtet sind die einzelnen Werte der Zeitreihe Realisationen $x_0, x_1, ..., x_n$ der Zufallsvariablen $X_0, X_1, ..., X_n$ mit stochastischen Eigenschaften, wie z.B. Varianz μ, Mittelwert δ und möglicherweise gemeinsamer Verteilung.
Ein populäres Verfahren ist ARIMA und setzt sich aus dem auto-regressiven Teil (AR), dem gleitendem Mittel (MA) und dem Integrations Teil (I) zusammen. Ohne weiter auf die anderen Bestandteile einzugehen ist der autoregressive Teil durch

$$X_t = \alpha_1 X_{t-1} + ... + \alpha_p X_{t-p} + \epsilon_t, t \in \mathbb{Z} \tag{1}$$

definiert. Hier werden für die Vorhersage der Variable zum Zeitpunkt t die vergangenen Variablen bis inkl. Zeitpunkt $t-p$ genutzt. $alpha_i$ sind dann Parameter, die den jeweiligen Zeitpunkt gewichten. Die Anzahl der Werte aus der Vergangenheit, welche für die Vorhersage genutzt werden, wird auch oft *Lag* genannt. Die Definition des *AR-Prozesses* ist auch für die neuronalen Architekturen relevant, da beispielsweise das Neural Network Auto-Regressive (NNAR), welches im späteren Verlauf der Arbeit behandelt wird, große Ähnlichkeit hat. [Sch01]
Es ist außerdem wichtig zu erwähnen, dass diese mathematischen Modelle nur sinnvoll Parameter schätzen können, falls gewisse Kriterien gelten. Beispielsweise ist bei der Zeitreihe in Abb. 1 kein Trend zu erkennen. Gäbe es zudem keine Saisonalität, wäre die Zeitreihe stationär und das o.g. Modell könnte angewendet werden.
Des Weiteren hat eine Zeitreihe meist eine gewisse Periodizität. So heizen die Menschen im Sommer meist weniger als im Winter. Dies lässt sich bereits leicht anhand von einer Visualisierung (wie in Abb. 1 zu sehen ist) entdecken.
Abgesehen von der Analyse des Graphen lassen sich die Autokorrelationsfunktion und weitere Bibliotheken anwenden, um mehr über die Zeitreihe zu erfahren. Die Autokorrelationsfunktion

berechnet den linearen Zusammenhang zwischen vergangenen Beobachtungen einer Zeitreihe zum jeweiligen Tag. So zeigt Abb. 2 die Anwendung der genannten Verfahren. Es ist deutlich sichtbar, dass der erste und zweite Tag vor der Vorhersage großen Einfluss auf den zu vorhersagenden Wert haben. Dies zeigt bereits einen ersten Hinweis darüber, dass für den Lag ein Wert von zwei ein guter Anhaltspunkt ist. Des Weiteren ist eine wöchentliche Saisonalität zu beobachten, was am sechsten und siebten Tag erkenntlich wird. Hier zeigt die Autokorrelationsfunktion einen Einfluss im negativen Wertebereich. Deshalb ist eine zusätzliche Beobachtung aus diesem Zeitfenster relevant für die Vorhersage des nächsten Tages.
Hierfür gibt es Variationen des o.g. Verfahrens, wie z.B. Seasonal Auto-Regressive Integrated Moving Average (SARIMA), welches u.a. abgesehen vom Lag Parameter (in den meisten Fällen ist das der Parameter p) einen Saisonalitätsparameter m, welcher die Periodendauer der Saisonalität bezeichnet und einen Parameter P, welcher die Anzahl an vergangenen „m-lags“ steuert, hat. Ohne genauer auf die Einzelheiten der jeweiligen Formeln einzugehen, ist es wichtig zu erwähnen, dass das gleiche Verfahren in den Variationen der später aufgezeigten neuronalen Modelle angewendet wird.
Um zuletzt die Saisonalität zu gewährleisten, ist es möglich die Differenzen der jeweiligen Tagesdaten zum Vorgänger zu betrachten. So kann mithilfe des i Parameters sowohl in ARIMA als auch in SARIMA und deren Variationen der Grad der Differenz bestimmt werden. Um zu überprüfen, ob ein gewisser Grad ausreichend für eine Stationarisierung einer Zeitreihe ist, kann der sog. „Dickey-Fuller-Test“, welcher im Python Paket statsmodels enthalten ist, durchgeführt werden. [Sch01]
Außerdem gibt es, abgesehen von den gewöhnlichen Differenzen, weitere Möglichkeiten eine instationäre Zeitreihe zu behandeln. So gibt es den sog. „Box-Jenkins-Ansatz“, welcher auch Potenzen der einfachen Differenzierung und ebenfalls saisonale Differenzen und deren Potenzen nutzt. Einige dieser Transformationen sind ebenfalls direkt über die Modellparameter der Modelle im statsmodels Python Paket konfigurierbar. [Sch01]
Eine weitere Limitierung der mathematischen Modelle ist, dass es standardmäßig ohne weiteres nicht möglich ist exogene Variablen zu berücksichtigen. Wenn z.B. der Zielwert (wie die Heizleistung von Haushalten) abhängig von anderen exogenen Variablen (wie z.B. der Außentemperatur und Jahreszeit) ist, so ist eine Vorhersage der Zukunftswerte präziser, wenn diese Attribute mitberücksichtigt werden. Für solche Fälle gibt es eine Variation der o.g. Modelle, wie z.B. *SARIMAX*, welche die Integration exogener Variablen ermöglicht und somit das Ergebnis potenziell verbessern kann.
Eine wichtige Analogie zu den später behandelten neuronalen Netzen ist also die Handhabung der (rohen) Daten, die Wahl der korrekten Lag und saisonalen Lag Parameter.

1.5 Datenaufbereitung für neuronale Netze

Im Gegensatz zu den genannten klassischen Verfahren, benötigt der Datensatz weniger Anpassungen, wenn er für ein neuronales Modell aufbereitet wird. Grund hierfür ist, dass die hohe Anzahl der Parameter und die nicht-linearen Aktivierungsfunktionen sehr komplexe Zeitreihen modellieren können.
Anders als bei z.B. ARIMA oder Extreme Gradient Boosting (XGB), (ein bekanntes Verfahren des maschinellen Lernens für die Klassifikation und Regression, welches jedoch auch für Zeitreihen angewendet werden kann), können problemlos mehrere Datenpunkte inklusive aller Attribute dem Modell übergeben werden.
Hierfür wird das sog. „Sliding Window“ Verfahren angewendet. Dabei wird aus einer sortierten Menge M mit N Elementen eine Eingabelänge $input\_width$ und eine Ausgabelänge

output_width definiert. Abb. 3 zeigt den Datensatz exemplarisch als Kacheln.

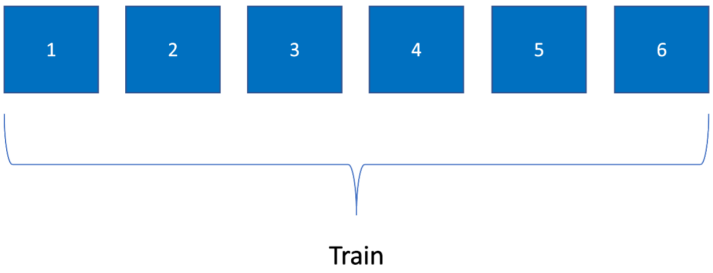
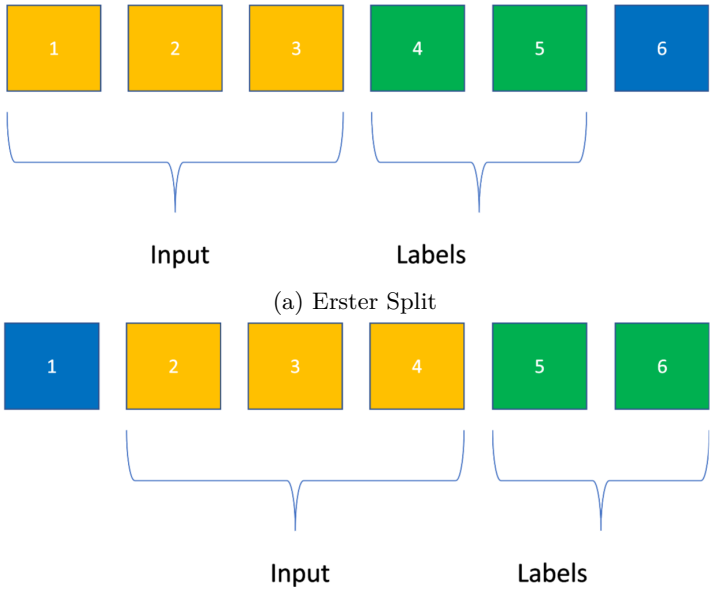


Abbildung 3: Trainingsdatensatz exemplarisch als Kacheln. Quelle: Eigene Darstellung

Anschließend werden Stücke mit der Länge $n = input\_width + output\_width$ ausgeschnitten, wobei immer am Anfang der sortierten Menge angefangen wird und das nächste Anfangselement anhand der Schrittweite *stride* ausgewählt wird. Abb. 4 zeigt, wie zwei Trainingsdatenpunkte aus den sechs Trainingsdaten hergestellt werden. Dabei gilt $N = 6$, $stride = 1$, $input\_width = 3$, $output\_width = 2$ und somit $n = 5$.



(a) Erster Split

(b) Zweiter Split

Abbildung 4: Visuelle Repräsentation des Sliding Window mit stride = 1, input_width = 3 und output_width = 2. Quelle: Eigene Darstellung

Abschließend werden bei den zu vorhersagenden Tagen (in der Abb. „Labels“) alle Features bis auf die Zielvariable entfernt. Abb. 5 zeigt, welche Features jeweils bei den jeweiligen Zeitschritten (in diesem Fall Tagen) vorhanden sind. Dabei zeigen die schwarzen Quadrate die Zielvariable.

Da die Zeitreihe immer wieder fehlende Werte in regelmäßigen Zeitschritten aufweist, muss dies speziell gehandhabt werden. Eine Möglichkeit für den Umgang mit Lücken ist die Entfernung von den jeweils sechs fehlenden Tagen. Dann ist jedoch die zeitliche Abfolge der Tagesdaten gestört, sobald die resultierende Zeitreihe wieder konkateniert wird. So sei die Zeitreihe $X = (x_1, x_2, x_3, ..., x_n)$ die Abfolge von Tagesdaten. Wenn nun z.B. x_2 entfernt wird, so folgt nach x_1 unmittelbar x_3. Da die Trainingsdaten aber nur zum Teil gestörte Abfolgen

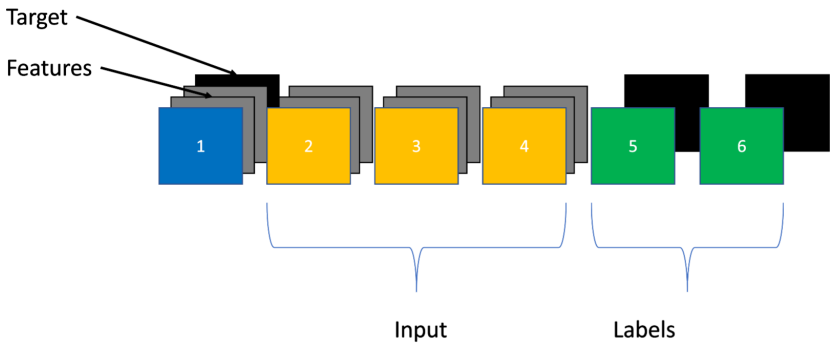
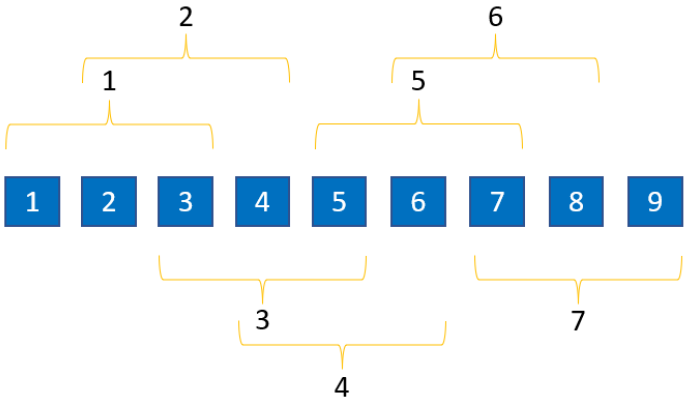


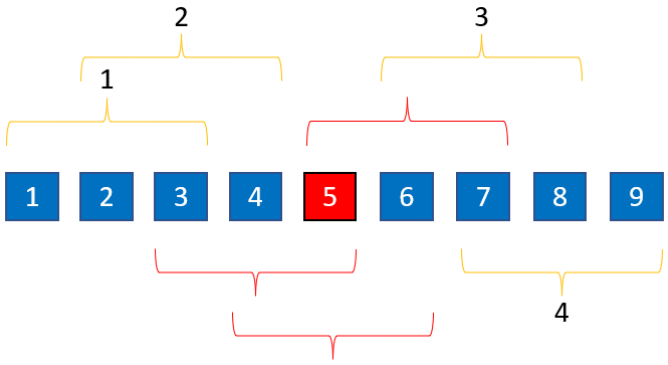
Abbildung 5: Featuretrennung bei den jeweiligen Eingabe- bzw. Ausgabezeitschritten. Quelle: Eigene Darstellung

beinhalten (da zum Anfang eine lückenlose Folge von 389 Tagen existiert), wird das Modell größtenteils mit widersprüchlichen Daten trainiert. Denn falls das Modell beim Training eine Beziehung zum Tag x_{t-i} in Hinsicht auf den zu vorhersagenden Tag x_t zieht, so ist diese Beziehung nicht mehr gültig, sobald der Tag x_{t-i} durch einen anderen Tag wie x_{t-i-1} ersetzt wird.
Am Beispiel der zugrundeliegenden Zeitreihe wird bei einem potenziellen Modell der Lag $input\_width = 2$ gewählt. Sei der vorherzusagende Tag ein Samstag, so nimmt das Modell die Werte vom Donnerstag und Freitag, um den gewünschten Tag vorherzusagen. Offensichtlich entsteht ein fehlerhafter Trainingsdatenpunkt, wenn statt dem Donnerstag (Tag $x_{t-i}, i = 2$) der Mittwoch aufgefüllt wird (Tag x_{t-i-1}).
Eine weitere Möglichkeit zur Handhabung der Lücken, die keinen Wert für die Zielvariable und andere wichtige Attribute haben, ist die (lineare) Interpolation. Jedoch ist das Problem, welches dabei entsteht, die systematische Verzerrung der Trainingsdaten, da die aufgefüllten Lücken keine echten Informationen enthalten.
Deshalb bleibt als einzige Möglichkeit, die Lücken aus dem Trainingsdatensatz zu entfernen, jedoch die restlichen Teilfolgen nicht miteinander zu konkatenieren. Sei $X = (x_1, x_2, x_3)$ eine zeitlich geordnete Folge der Zielvariable. Dann entstehen durch die Entfernung von x_2 die beiden Teilfolgen $X_1 = (x_1)$ und $X_2 = (x_3)$. Die ursprüngliche Folge X lässt sich in einen Trainingsdatenpunkt umwandeln, indem die Variablen x_1 und x_2 als Vergangenheitswerte und x_3 als Zielvorhersage verwendet werden. Offensichtlich ist hier der Lag $input\_width = 2$ und somit lässt sich kein Trainingsdatenpunkt aus den Teilfolgen X_1 und X_2 mehr herstellen.

Im Fall der zugrundeliegenden Zeitreihe sind immer 24 Tage vollständig gegeben und darauf folgen sechs Tageslücken. Sei $X = (X_1, X_2, X_3)$ eine Folge von Folgen und sei $X_1 = (x_0, ..., x_{24})$ die Folge von 24 vollständig gegebenen Tagen, $X_2 = (x_{25}, ..., x_{30})$ sechs Tageslücken und $X_3 = (x_{31}, ..., x_{54})$ eine weitere Folge von 24 vollständig gegebenen Tagen. Dann entstehen durch die Entfernung von X_2 die impliziten Teilfolgen X_1 und X_3, welche jeweils 24 Elemente enthalten. Bei einem Lag von $input\_width = 14$, einer Vorhersageperiode von $output\_width = 6$ muss ein Trainingsdatenpunkt genau aus $input\_width + output\_width = 20$ Tagen bestehen. Um eine verzerrte Folge zu verhindern, darf keine Konkatenation der Teilfolgen X_1 und X_3 stattfinden.Demnach müssen einzeln Trainingsdatenpunkte aus jeweils X_1 und X_3 mit dem o.g. Sliding Window Verfahren extrahiert werden. Bei einer Anzahl von $|X_1| = |X_3| = 24$ Tagesdaten lassen sich somit „nur“ $2 * [|X_1| - (input\_width + output\_width + 1] = 2 * (24 - 20 + 1) = 10$ Trainingsdatenpunkte erstellen. Verglichen mit der Anzahl der Trainingsdatenpunkte, die durch das Sliding Window Verfahren nach einer Interpolation der Trainingsdatenlücken entstehen,



(a) Ursprüngliche Menge der Trainingsdatenpunkte



(b) Menge an Trainingsdatenpunkten nach Entfernung von dazwischenliegendem Tag ohne Konkatenation

Abbildung 6: Veranschaulichung vom Trainingsdatenverlust bei korrekter Extrahierung von Trainingsdatenpunkten aus gesamter Datenmenge mit Lücken. Quelle: Eigene Darstellung

ist das sehr wenig. Denn hier wären insgesamt

$$|X| - (input\_width + output\_width) = |X_1| + |X_2| + |X_3| - (input\_width + output\_width) + 1$$
$$= 24 + 6 + 24 - 20 + 1 = 35$$

Trainingsdatenpunkte möglich.
Da also die Interpolation als auch die ausnahmslose Entfernung der Lücken mit anschließender Konkatenation die Trainingsdaten verfälschen, bleibt nur der zuletzt gewählte Ansatz. Die einzige Möglichkeit, um die Anzahl der gewonnenen Trainingsdatenpunkte zu erhöhen, ist die Anzahl der vergangenen als auch zu vorhersagenden Tage zu reduzieren.
Zuletzt werden die aufbereiteten Datenpunkte in Tensoren geladen, sodass sie später dem Model zum Training und zur Evaluation übergeben werden können.

2 Grundlagen neuronaler Architekturen für Zeitreihenvorhersage

2.1 Das Perzeptron

Anders als die mathematischen Modelle können neuronale Modelle Nichtlinearitäten und komplexe Zusammenhänge sehr gut abbilden. Deshalb ist die Stationarität der Zeitreihe nicht zwingend notwendig für das Training eines neuronalen Netzwerks, kann das Ergebnis jedoch trotzdem verbessern.
Die neuronalen Netze und das sog. Perzeptron orientieren sich an den menschlichen Nervenzellen. Diese haben sehr viele Nervenverbindungen und produzieren zu einer jeweils einzigartigen Kombination aus Eingabeimpulsen einen oder mehrere Ausgabeimpulse. Demnach ist es durch viele nacheinander geschaltete Nervenzellen möglich, vielfältige Aufgaben, wie Kommunikation, Muskelsteuerung, Sprache, uvm. zu erlernen. [Gé19]
Im Bereich des maschinellen Lernens ahmt das sog. Perzeptron die Eigenschaften und Funktionsweise einer Nervenzelle grob nach. Hiermit wird ein Netzwerk mit nur einem einzigen Neuron bezeichnet. Abb. 7 zeigt, wie die Verknüpfung zwischen Ein- und Ausgabe umgesetzt ist. Im Sinne einer Zeitreihe kann die Abfolge von Eingabedaten die einzelnen Zeitreihenwerte, wie z.B. die Abfolge von Temperaturen, Spritpreisen oder Aktienpreisen, enthalten. Jedes Neuron ist mit allen Ausgaben der vorherigen Schicht verknüpft. In diesem Fall entspricht die erste Schicht der Eingabeschicht und deswegen sind alle Eingaben mit dem darüberliegendem Neuron verbunden. [Gé19]

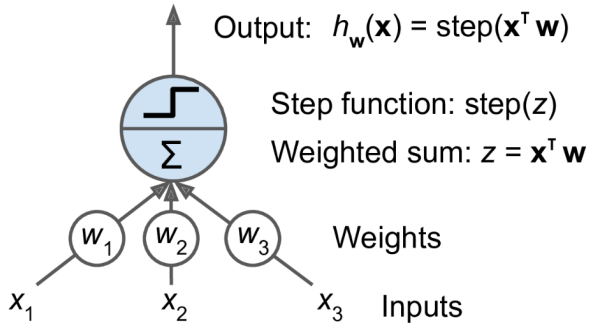


Abbildung 7: Ein einzelnes Perzeptron. Quelle: [Gé19]

In Abb. 7 werden die Gewichte als w_1, w_2, w_3 und die Eingaben als x_1, x_2 und x_3 definiert. Die gewichtete Summe lässt sich ebenfalls für alle Batchelemente (also alle in einem vorwärts Durchlauf zu verarbeitende Datenpunkte) berechnen, indem Matrixweise gerechnet wird, wie in Gleichung 2 zu sehen ist.
Üblicherweise bestehen neuronale Architekturen aus mehreren sog. Schichten. Eine Schicht enthält wiederum mehrere einzelne Neuronen. Die Eingabeschicht eines neuronalen Netzwerks sendet die Eingabemerkmale an die darüberliegende Schicht. Sei die Eingabe $e = (x_1, x_2, x_3, ..., x_i)$, wobei i die Anzahl an Eingabemerkmalen ist, dann gibt es für jedes Neuron der darüberliegenden Schicht die Gewichte $(w_1, w_2, w_3, ..., w_i)$, welche die jeweiligen Eingänge gewichten. Für den Fall, der in Abb. 7 skizziert ist, hat die Schicht nur ein einziges Neuron.

$$h_{W,b}(X) = \Phi(XW + b) \tag{2}$$

2.2 Aktivierungsfunktionen

Dabei ist Φ die sog. Aktivierungsfunktion, welche die nicht lineare Transformation ermöglicht. Ohne die Aktivierungsfunktion können auch nacheinander geschaltete Schichten in eine große lineare Gleichung zusammengelegt werden und das gesamte Netz fällt somit zu einer Schicht zusammen. [Gé19]
In diesem Fall wird eine der einfachsten Aktivierungsfunktionen, die sog. „Einheitssprungfunktion“ benutzt, welche mit

$$f(x) = \begin{cases} 1, & \text{falls } x \geq 0 \\ 0, & \text{falls } x < 0 \end{cases}$$

definiert ist und in Abb. 8a zu sehen ist. Sie „aktiviert“ das Neuron bei positiver Eingabe. Weitere Aktivierungsfunktionen sind in Tabelle. 1 zu sehen. Wichtig hierbei ist die sog. *ReLU* Funktion, welche sehr oft in neuronalen Netzen seine Verwendung findet. Das Problem bei der Funktion ist jedoch, dass die Ableitung als auch der Funktionswert für alle $x < 0$ Null ist. Deshalb kann es passieren, dass viele Neuronen beim Gradientenabstieg nicht mehr aktualisiert werden, sobald die Eingabe für die Funktion negativ ist. Häufig wird dieses Phänomen auch das „Absterben von Neuronen“ genannt.
Sowohl die *SELU* als auch die *Leaky ReLU* wurden speziell dafür entwickelt, um die Nachteile der *ReLU* zu kompensieren. In der Praxis erweist sich jedoch die neue *Swish* Funktion am effektivsten, was in den späteren Tests in Kap. 4.1 zu sehen ist. [Xu20] [Ram17] [Kla17]

Tabelle 1: Weitere Aktivierungsfunktionen. Quellen: [Nwa18], [Ram17], [Kla17], [Xu20]

Name	Funktion	Kommentar
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	-
Tanh	$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1-e^{-2x}}{1+e^{-2x}}$	-
ReLU	$Relu(x) = max(0, x)$	-
GelU	$Gelu(x) = x\Phi(x)$	Φ ist Verteilungsftk. der Gaußverteilung
Swish	$swish(x) = x\sigma(\beta x)$	in TensorFlow gilt $\beta = 1.0$
Leaky ReLU	$leakyrelu(x) = max(\alpha x, x)$	α ist normalerweise 0.01
SELU	$selu(x) = \begin{cases} \lambda x, & \text{falls } x > 0 \\ \lambda\alpha(e^x - 1), & \text{falls } x \leq 0 \end{cases}$	in TensorFlow gilt $\alpha = 1.67326324$ und $\lambda = 1.05070098$

Prinzipiell hängt also die Wahl der Aktivierungsfunktion von der Aufgabenstellung, Anzahl der Neuronen, Anzahl der Schichten und der Art der verwendeten Schichten ab. Trotz des

theoretischen Vorteils der neueren Aktivierungsfunktionen gegenüber der *ReLU* Funktion, kann die *ReLU* Funktion in manchen Szenarien vorteilhaft sein.
Nicht nur deshalb, sondern auch weil im Verlauf der Arbeit viele verschiedene Architekturen evaluiert werden, müssen alle Kombinationen für jedes Modell getestet werden, damit ein optimales Ergebnis sichergestellt werden kann. Deshalb sind die in Tab. 1 gelisteten Aktivierungsfunktionen alle in der Gittersuche enthalten.

Ausgabeschicht

Der vollständigkeitshalber werden die möglichen Aktivierungsfunktionen für die Ausgabeschicht genannt. Je nachdem, welche Aufgabe gelöst werden soll, werden unterschiedliche Aktivierungsfunktionen (oder auch keine) in der Ausgabeschicht verwendet. So wird bei „multi-class“ Klassifikation (also einer Klassifikation bei der einem Datenpunkt eine eindeutige Klasse zugewiesen wird) die Softmax Funktion verwendet, die durch

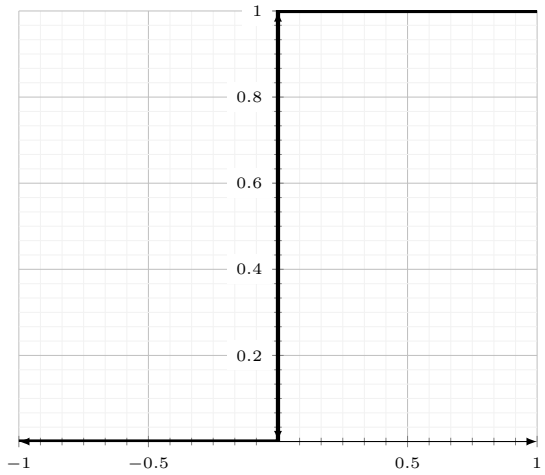
$$f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{für } spacei = 1, 2, \ldots, K$$

definiert ist. Dabei sind z_i die jeweiligen rohen Ausgaben der Neuronen. Das Ergebnis der e-Funktion für das jeweilige z_i wird durch die Summe aller e-Funktionswerte geteilt, was erklärt, warum das Ergebnis aller Softmax Ausgaben sich zu eins addiert. Hat eine Ausgabeschicht z.B. drei verschiedene Neuronen, welche die Wahrscheinlichkeit für die jeweilige Klasse repräsentieren, so berechnet die Softmax aus den rohen Ausgaben der Neuronen Wahrscheinlichkeiten, welche sich zu eins addieren. Deshalb wird diese Funktion in Szenarien verwendet, bei denen sich die jeweiligen Klassen gegenseitig ausschließen. Es wird nur die Klasse mit der höchsten Wahrscheinlichkeit klassifiziert. [Gé19]
Wenn hingegen mehrere Klassen einem Datenpunkt zugewiesen werden können, wird die Sigmoide Funktion, die durch

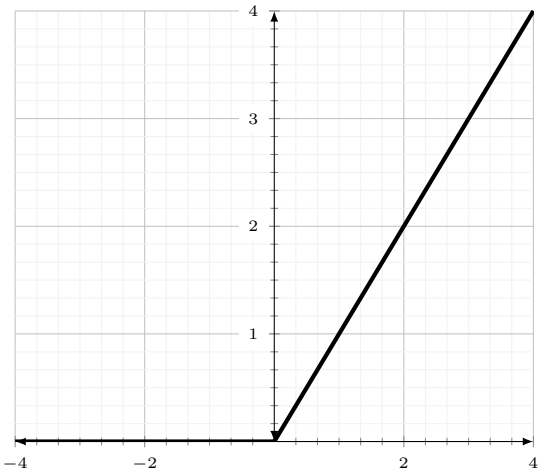
$$f(z) = \frac{1}{1 + e^{-z}}$$

definiert ist, benutzt. Diese berechnet die Wahrscheinlichkeiten unabhängig von den Ausgaben der anderen Neuronen. Die Wahrscheinlichkeiten können also separat betrachtet werden und es wird ein Schwellwert festgelegt, ab welchem dem Datenpunkt eine jeweilige Klasse zugewiesen wird. Dies wird auch als „multi-label“ Klassifikation bezeichnet. [Gé19]
Bei der Regression hingegen wird in der Ausgabeschicht meist keine Aktivierungsfunktion verwendet. Soll ein gewisser Wert vorhergesagt werden, können zwar in den verborgenen Schichten Aktivierungsfunktionen verwendet werden, jedoch verzerren manche Aktivierungsfunktionen die Ausgabe (wie z.B. die ReLu Funktion), sodass z.B. negative Werte nur auf 0 abgebildet werden.

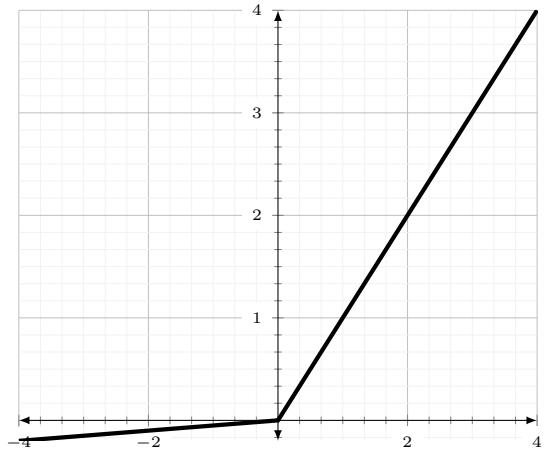
Zuletzt gibt es noch den sog. Bias, welcher als b definiert wurde. Dieser ermöglicht eine zusätzliche Addition eines Wertes, nachdem die gewichtete Summe der Ausgaben der letzten Schicht produziert wurde. Damit kann selbst ein niedriges Ergebnis der gewichteten Summe eine potenziell hohe Ausgabe der Aktivierungsfunktion bewirken. [Gé19]
Mithilfe der o.g. Aktivierungsfunktionen und Neuronen können somit mathematische Modelle, wie z.B. der AR-Prozess nachgebildet werden. Zusätzlich dazu ermöglichen die nicht linearen Aktivierungsfunktionen und der Einsatz mehrerer Schichten eine Modellierung eines nicht linearen Problems. [Gé19]
Im folgenden Kapitel werden Tensoren erklärt, welche eine Grundlage für die Rechenoperationen



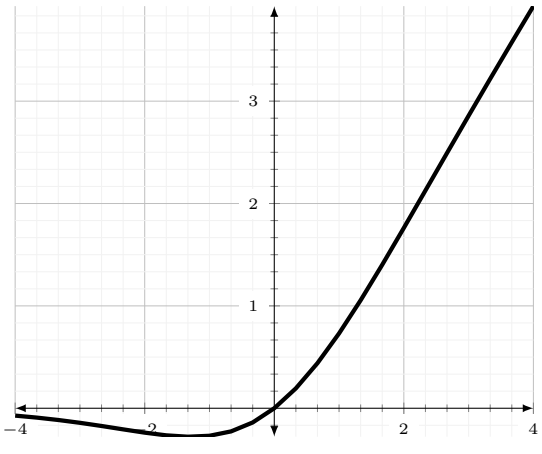
(a) Treppenfunktion



(b) ReLu Funktion



(c) Leaky ReLu Funktion mit $\alpha = 0.05$



(d) Swish Funktion mit $\beta = 1.0$

Abbildung 8: Aktivierungsfunktionen. Quelle: Eigene Darstellung der Funktionen

der heutigen neuronalen Netze darstellen. Anschließend wird exemplarisch anhand des bereits gezeigten Datensatzes und einer einzelnen verborgenen Schicht eine Zeitreihenvorhersage demonstriert.

2.3 Tensoren und Daten

In Kap. 1.5 wurde bereits gezeigt, wie aus der Grundmenge der Daten einzelne Trainingsdatenpunkte extrahiert werden können. Da die Daten jedoch nicht in einem willkürlichen Format vorliegen, wie z.B. einer Pandas *DataFrame* (dies ist eine gängige Klasse zur Darstellung, Analyse und Bearbeitung von tabellarischen Daten in Python), müssen diese zuerst in einen Tensor umgewandelt werden.
Ein Tensor ist ein multi-dimensionales Array, welches alle Informationen der Trainingsdaten, eines Trainingsdaten Batch (dies ist eine Sammlung von mehreren Trainingsdatenpunkten) oder eines einzelnen Trainingsdatenpunkts in einem numerischen Format hält.
So ist z.b. ist ein Tensor mit Rang 0 effektiv ein Skalar, ein Tensor mit Rang 1 ein Vektor und ein Tensor mit Rang 2 eine Matrix. [Dul10]

Am Beispiel der vorliegenden Zeitreihe und der zugrundeliegenden Problemstellung wird z.B. ein Lag von $p = 14$ gewählt. Sei $X = (x_{t-1}, x_{t-2}, ..., x_{t-p})$ wobei $x_i \in \mathbb{R}^n, n \in \mathbb{N}$ ein Trainingsdatenpunkt, der die Daten der vergangenen p Tage hält. So sind die $x_i \in \mathbb{R}^n$ mehrdimensionale Vektoren, welche jeweils alle Attribute für einen jeweiligen Tag beinhalten.
Hat z.B. ein Tag die Werte $SystemBuy = 62{*}10^7$, $SystemSell = 13{*}10^7$ und $margin = 49{*}10^7$, so kann dieser Tag auch als Vektor

$$\vec{x_i} = \begin{pmatrix} 62 * 10^7 \\ 13 * 10^7 \\ 49 * 10^7 \end{pmatrix}$$

definiert werden.
Aus Sicht der Informatik kann dies auch als ein Array der Form

```
tag_t_1 = [62*10^7, 13*10^7, 49*10^7]
tag_t_2 = [112*10^7, 14*10^7, 98*10^7]
lags = [tag_t_1, tag_t_2]
```

geschrieben werden. Dabei wird die Anzahl der Indizes des Tensors auch Rang genannt. In diesem Fall ist der Tensor Rang zwei und stellt effektiv eine quadratische Matrix dar. In *tensorflow* lassen sich Tensoren über Hilfsfunktionen direkt aus nativen python oder numpy (das ist eine Bibliothek für wissenschaftliches Rechnen) Arrays herstellen.
Im Rahmen dieser Arbeit wurde ein Verfahren entwickelt, welches das in Kap. 1.5 genannte Problem behebt, indem es Trainingsdatenpunkte nur aus vollständigen Teilmengen der Daten generiert. Zudem lädt das Verfahren die resultierenden Trainingsdaten mithilfe von Hilfsmethoden der Bibliothek *tensorflow* in Tensoren. Eine Limitierung bei der Handhabung mit Tensoren ist, dass deren Rang und die Dimensionen für die jeweiligen Indizes wohldefiniert sein müssen. Dies ist der Fall, da, abgesehen von der offensichtlichen Notwendigkeit für Speicherallokation, die Modelle in den jeweiligen Schichten zum Eingabetensor passen müssen. Denn, wie bereits in Kap. 2 erklärt, muss beispielsweise jedes künstliche Neuron einer einfachen Schicht je ein Gewicht pro Eingabezahl haben.
Die Inhalte zweier Tensoren können, solange die Anzahl der Ränge und Dimensionen in den Indizes übereinstimmt, addiert werden. Falls es gewünscht ist, Eingaben verschiedener „Formate“ und somit nicht kompatibler Tensoren zusammen zu verarbeiten, müssen andere Verfahren, wie das in Kap. 3.4, eingesetzt werden.

2.4 Training mithilfe von Verlustfunktion, Backpropagation und sog. Optimierern

Damit das neuronale Netz seine Gewichte anpassen kann, um zuletzt den Fehler der Ausgabe zu minimieren, benötigt es einer Verlustfunktion. Diese berechnet den Fehler von der Vorhersage des Modells zu der Zielvariable.
Eine der bekanntesten Verlustfunktionen, die ihre Verwendung bei Regressionsproblemen findet, ist sowohl der Mean Absolute Error (MAE), als auch der Mean Squared Error (MSE). Diese beiden Funktionen sind jeweils durch

$$mae(x, y) = \sum_{i=1}^{D} |x_i - y_i| \ ,$$
$$mse(x, y) = \sum_{i=1}^{D} (x_i - y_i)^2$$

definiert. Dabei sind x die vorhergesagten Werte, y die tatsächlichen Werte und D die Dimension der Ausgaben. Beispielsweise ist die Dimension der Ausgabe $D = 2$, wenn das Modell zwei Tage und pro Tag nur die Zielvariable *margin* vorhersagt.
Dies ist exemplarisch in Abb. 9 dargestellt. Hierbei gibt es zum einen die Besonderheit, dass sowohl die Ausgabe des Modells als auch die jeweiligen Zielvariablen in Tensoren dargestellt sind.

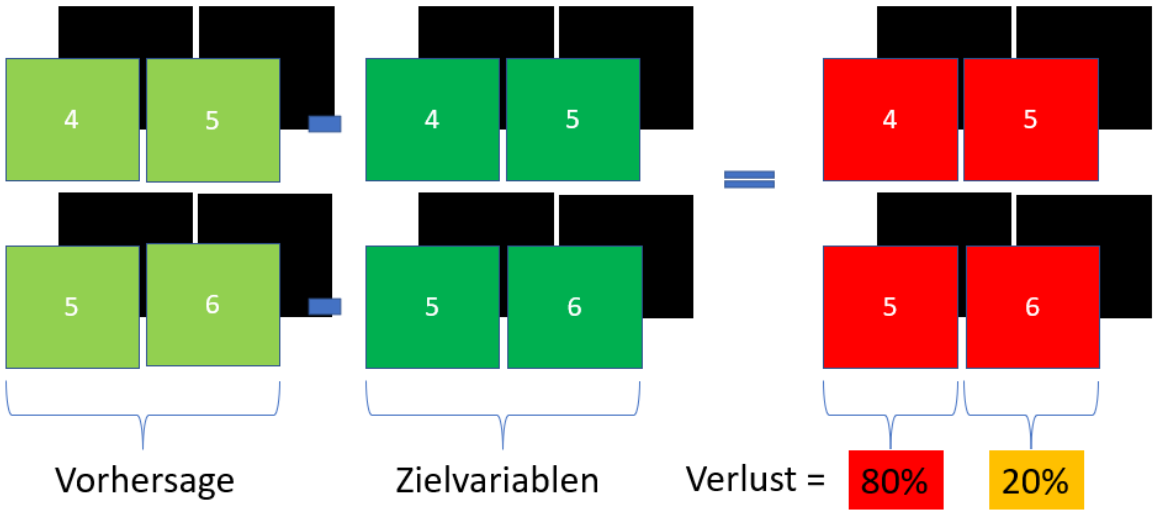


Abbildung 9: Angepasste Verlustfunktion. Quelle: Eigene Darstellung

Zum anderen ist zu sehen, dass nach der Bildung der Differenz eine gesonderte Skalierung durchgeführt wird. Dies ist ein experimenteller Versuch das Training des Modells zugunsten der Auswertung und Scorings des Wettbewerbs zu gestalten. Hierbei wird auf den ersten zu vorhersagenden Tag eine höhere Gewichtung als auf die restlichen Tage gelegt. Die angepasste Verlustfunktion ist also durch

$$mse\_gewichtet(x, y) = \sum_{i=1}^{D} w_i(x_i - y_i)^2 \ , w_i \in (0, 1)$$

definiert und bei einer 6-Tages-Vorhersage gilt bei den in der Arbeit durchgeführten Tests immer $\vec{w} = (0.5, 0.125, 0.125, 0.125, 0.125)^T$.
Somit hat das Modell beim Training eine wohldefinierte Verlustfunktion und kann damit schrittweise die Ableitung der Verlustfunktion und die Ableitung der davor geschalteten Aktivierungsfunktionen berechnen. Abschließend können mithilfe der Kettenregel und den partiellen Ableitungen die jeweiligen Gradienten bestimmt werden, sodass die davorliegenden Gewichte angepasst werden können. [SUT86]
Das beschriebene Verfahren heißt *Backpropagation* und benötigt am Anfang die Ausgaben aller Neuronen, die in diesem Netz eingesetzt werden. Deshalb ist der erste Schritt (der sog. *Forward Pass*), die Resultate für alle Neuronen und deren Aktivierungsfunktionen für eine spezifische Eingabe zu berechnen. Der Algorithmus fängt abschließend von hinten an (daher sein Name) die Gewichte und Bias Terme entsprechend dem jeweiligen Gradienten anzupassen. [SUT86]
Hierfür wird klassischerweise das Gradientenabstiegsverfahren (auch als *SGD* bezeichnet)

verwendet, mithilfe dessen eine schrittweise Annäherung an die optimale Lösung möglich ist. Abb. 10 zeigt den Graphen einer Verlustfunktion.

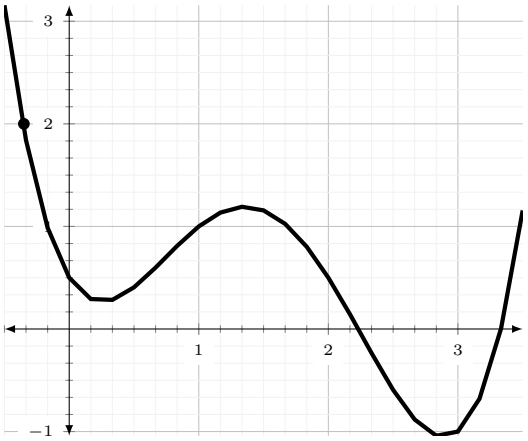


Abbildung 10: Verlustfunktion für ein gegebenes Gewicht. Quelle: Eigene Darstellung

Wie zu sehen ist, hat die Funktion mehrere Minima. Sei der Wert der Verlustfunktion $y = 2$ für einen beispielhaften Parameter $x = -0.35$ (was z.B. der Bias vor der Verlustfunktion sein könnte), dann ist durch die Ableitung der Verlustfunktion an der Stelle x eine Steigung berechenbar. Mithilfe des errechneten Gradienten ist nun eine Anpassung des Parameters möglich, sodass die Verschiebung des Parameters einen niedrigeren Verlust beim nächsten Vorwärtsdurchlauf (und damit auch einer Vorhersage) ergibt. Dabei wird der Parameter nicht um einen konstanten Wert verändert, sondern es wird eine sog. „Lernrate“ μ eingesetzt, um den Gradienten an der jeweiligen Stelle x zu skalieren und das Ergebnis auf den ursprünglichen Parameter x zu addieren. Hiermit lässt sich ebenso erkennen, dass sowohl die Verlustfunktion als auch die Aktivierungsfunktionen in einem neuronalen Netz differenzierbar sein müssen.
Ein Problem, welches bei dem Verfahren auftritt, ist, dass nur selten das globale Minimum erreicht wird. Grund hierfür ist, dass der Gradient im Punkt $x = 0.5$ nahezu Null ist und die Änderung des Parameters mit einer konstanten Lernrate in diesem Fall entsprechend gering ist. Da das neuronale Netz üblicherweise so lange trainiert wird, bis der Fehler auf den Trainings- bzw. Validierungsdaten nicht mehr signifikant fällt, würde das Training in diesem Fall in einem lokalen Minimum beendet werden.
Zur heutigen Zeit gibt es Verbesserungen des beschriebenen Verfahrens, wie z.B. den *Adam* und *RMSProp* Optimierer. Diese verfügen über weitere Mechanismen, wie z.B. das sog. *Momentum*, welches das Überwinden von lokalen Minima ermöglicht. So ist die Idee des Momentums eine gewisse Geschwindigkeit zu berechnen, je nachdem wie lange der Gradient fällt und wie stark die Steigung jeweils ist. Somit baut das Verfahren ein Momentum auf, sodass es nicht unmittelbar bei Erreichung eines Minimas die Änderungsrate reduziert und somit über das Minimum in ein potenziell tieferes Minimum (wie z.B. die Koordinate (4,5)) flüchten kann. [Kin14] [Rud16]
Die genannten Optimierer verfügen ebenfalls über eine einstellbare Lernrate als auch andere Parameter, welche z.B. die Stärke des Momentums steuern. Da in dieser Arbeit viele verschiedene Modellarchitekturen in Kombination mit anderen Hyperparametern, wie Aktivierungsfunktionen, Anzahl der Neuronen, etc. getestet werden und die Menge der Trainingsdaten ebenso variiert (aufgrund der Regeln des Wettbewerbs), wird im Laufe dieser Arbeit der *COCOB* Optimierer verwendet. Dieser hat keine konfigurierbare Lernrate und findet die Lern-

rate selbstständig. Er ist somit gut geeignet um schnell einen Überblick über die Qualität einer Gittersuche zu bekommen und die Änderungen an Modellarchitekturen entsprechend zu bewerten, da er in vielen Fällen eine zur Gegebenheit passende Lernrate errechnet und entsprechend während dem Training variiert. [Ora17]

3 Implementierung und Evaluation verschiedener Architekturen zur Zeitreihenvorhersage

Daten, Evaluation und Training

Wie bereits in Kap. 1.5 erklärt, ändert sich die Menge der Trainingsdaten, falls die Summe der beinhalteten Vergangenheitswerte als auch die der zu vorhersagenden Tage im Trainingsdatenpunkt variiert. Dies hat jedoch nicht zu bedeuten, dass das jeweilige Modell alle Vergangenheitswerte zur Vorhersage nutzt. So können, je nach Architektur, auch nur zwei Vergangenheitswerte vom Modell in die jeweiligen Schichten eingegeben werden, um eine Vorhersage zu produzieren.
Beispielsweise hat sich bei den Tests im Rahmen dieser Arbeit erwiesen, dass eine Änderung der Anzahl der Vergangenheitswerte von $input\_width = 7$ auf $input\_width = 14$ die Anzahl der Validierungsdaten um 43.75% verringert. Somit ergeben sich auch verschiedene Fehler trotz Evaluation des gleichen Modells. Deshalb wird in den folgenden und auch allen nachfolgenden Tests der Datensatz jeweils mit einer $input_width = 14$ generiert, um die Anzahl der Trainings- als auch Validierungsdatenpunkte gleich zu halten.
Konkret wird der Zeitraum vom 01. Oktober 2011 bis inkl. 12. April 2015 für das Training genutzt. Für die Validierungsdaten bleibt der Zeitraum vom 13. April 2015 bis zum 02. Dezember 2016. Durch Anwendung des in Kap. 1.5 beschriebenen Verfahrens mit Entfernung der Lücken und ohne Konkatenation resultierender Teilmengen, können 861 Trainingsdatenpunkte und 320 Validierungsdatenpunkte gewonnen werden. Dies ist ein übliches Verhältnis zur Aufteilung von Trainings- und Testdaten und beträgt in diesem Fall 72.90%.
Der Begriff Testdaten bezeichnet dabei normalerweise Daten, welche zur Evaluation nach dem Training benutzt werden und dabei weder für Hyperparametertuning noch für das Training benutzt werden. Da jedoch im Rahmen dieser Arbeit die Hyperparameter verschiedener Modelle hinsichtlich dieses Datensatzes optimiert werden, ist die Auswertung zu optimistisch. Trotzdem ist es gewünscht, dass das Modell kurz vor dem Vorhersagezeitraum möglichst präzise Vorhersagen liefert (und der Validierungszeitraum liegt immer genau vor einer jeweiligen Vorhersageperiode), weshalb im Laufe der Arbeit die Evaluation hinsichtlich des Validierungsdatensatzes getätigt wird. Eine weitere und etwas präzisere Testmöglichkeit ist die Kreuzvalidierung, welche jedoch wesentlich rechenintensiver ist und in Verbindung mit Zeitreihen komplexer umzusetzen ist.
Des Weiteren ist anzumerken, dass für alle folgenden Trainingsabläufe eine maximale Anzahl von 50 Epochen gesetzt ist und ein sog. „Early-Stopping“ mit $patience = 10$ angewendet wird. Dies bedeutet, dass das Modell höchstens 50-mal mit den gleichen Trainingsdaten wiederholt trainiert wird, jedoch sofort abgebrochen wird, sobald sich der Fehler auf dem Validierungsdatensatz nicht mindestens ein Mal innerhalb zehn aufeinanderfolgenden Trainingszyklen verbessert hat.

Mittelwertsprognose als Gütevergleich

Zwar ist es möglich mithilfe der Validierungsdaten das beste Modell bezüglich des niedrigsten Fehlers auszuwählen, jedoch ist unklar, ob die Vorhersagen besser als z.B. der Mittelwert sind. Aus dieser Sicht ist ein Modell trotz niedrigstem Fehler schlecht, sobald es nicht signifikant besser als der Mittelwert ist.
Der Mittelwert für die o.g. Validierungsdaten ist c.a. $\mu = 78367200$ und wird somit bei jeder Evaluation der folgenden Modelle zum Vergleich gezogen.

TensorFlow 2

Um die verschiedenen Architekturen effizient zu implementieren ohne dabei die grundlegenden Mechanismen der neuronalen Netze programmieren zu müssen, wird TensorFlow 2 eingesetzt. TensorFlow ist, zusammen mit PyTorch, eine der heutzutage bekanntesten Bibliotheken für die Entwicklung von neuronalen Netzen. Sie bietet Methoden zum Laden von Datensätzen, Hilfsmethoden für die Datenvorverarbeitung, verfügt über Implementierungen von allen gängigen neuronalen Schichten, Aktivierungsfunktionen und Optimierern. Darüber hinaus gibt es viele weitere Funktionalitäten, auf welche hier jedoch nicht näher eingegangen wird.
Wichtig ist, dass beide Frameworks mithilfe der Schnittstelle *CUDA* eine Hardwarebeschleunigung mit einer Grafikkarte ermöglichen. Die Wahl für diese Arbeit fällt auf Tensorflow, da in Tensorflow zum Teil einfache Modelle mit wenig Zeilen an Code programmiert werden können und das Training von Modellen keine so hohe Anzahl an Zusatzsyntax und Befehlen wir PyTorch benötigt.
Im nächsten Kapitel wird gezeigt, wie eine einzelne verborgene Schicht eine Zeitreihe vorhersagen kann und wie hoch der Fehler bei der Auswertung auf den beschriebenen Daten ist.

3.1 Vorhersage mit einer verborgenen Schicht

Sei $shape = (batch\_size, time, features)$ das Format eines Tensors, wobei $batch\_size$ die Anzahl der Dimensionen im ersten Index und gleichzeitig die Anzahl er Trainingsdatenpunkte in einem Batch ist, $time$ die Anzahl an vergangenen Datenpunkten ist und $features$ die Anzahl der Attribute für jeden Tagesdatenpunkt ist. Der Tensor hat somit Rang drei und hält $batch\_size$ Trainingsdatenpunkte, wobei jeder Trainingsdatenpunkt aus $time * features$ numerischen Werten besteht.
Wenn beispielsweise eine Batchgröße von 32 gewählt wird, $lag = 14$ (für die Generierung der Trainingsdatenpunkte) und die Anzahl der Features $features = |\{SystemBuy, SystemSell, margin\}| = 3$, dann hat der Tensor mit 32 Trainingsdatenpunkten (Batchgröße) das Format $shape_{tensor\_1} = (32, 14, 3)$. Falls für einen Tag genau ein Attribut vorhergesagt werden soll (wie z.B. $margin$), so muss das Modell den Eingabetensor in einen Tensor mit Format $shape\_out = (32, 1, 1)$ transformieren.

Abb. 11 zeigt eine Möglichkeit, wie der Eingabetensor transformiert werden kann, falls er drei vergangene Tage mit jeweils vier Attributen hält. Im ersten Schritt wird der Tensor von einem Rang 3 Tensor mit $shape = (32, 3, 3)$ zu einem Rang 2 Tensor mit $shape = (32, 3 * 3)$ transformiert. Anschließend (dies ist nicht mehr in der Abb. enthalten) wird der resultierende Tensor an eine Schicht mit nur einem Neuron weitergegeben, sodass der resultierende Tensor das Format $shape = (32, 1)$ hat. Um zuletzt das Format der gewünschten Ausgabe zu erhalten (es soll ein Tag vorhergesagt werden und davon nur eine Zielvariable), muss ein Rang künstlich

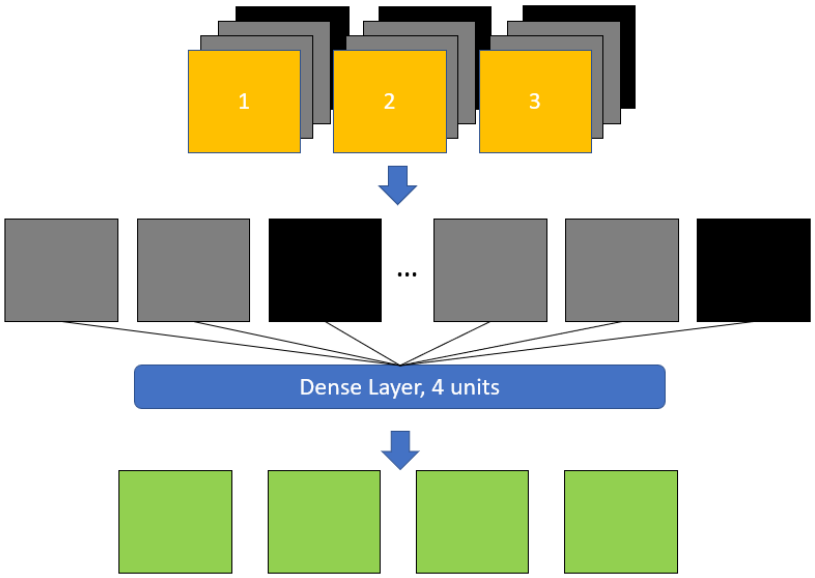


Abbildung 11: Eine verborgene Schicht. Der Batch wurde absichtlich ausgelassen. Quelle: Eigene Darstellung

erzeugt werden, sodass der resultierende Tensor $shape = (32, 1, 1)$ und somit das Format der Labels (das sind die Zielvariablen mit originalen Daten) erreicht.

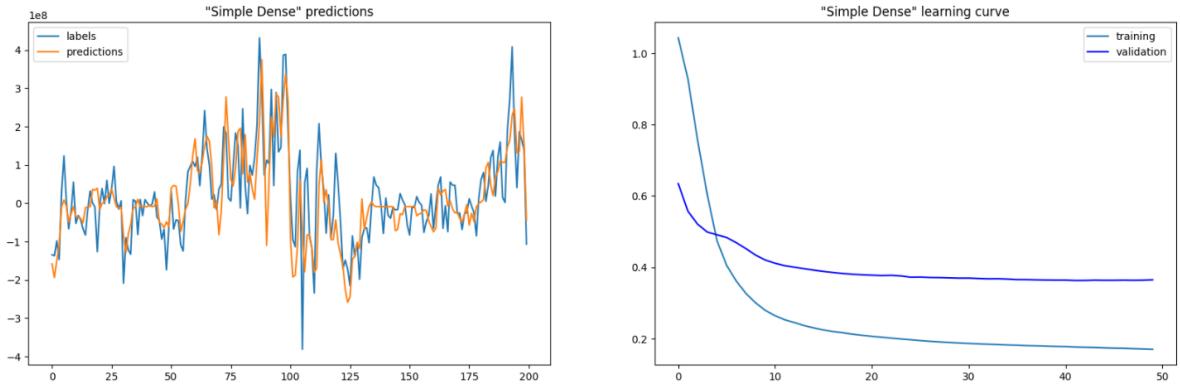


Abbildung 12: Vorhersage und Verlust im Verlauf mehrerer Trainingszyklen. $mae_{simple\_dense} \approx 59828492$. Quelle: Eigene Darstellung

Abb. 12 zeigt auf der linken Seite, wie das Modell nach dem Training die Validierungsdaten iterativ vorhersagt. Das Modell schafft es in den meisten Fällen eine zum Originalwert ähnliche Vorhersage zu tätigen, schlägt jedoch manchmal (wie zwischen Epoche 70 und 100 zu sehen) in die entgegengesetzte Richtung aus.
Auf der rechten Seite ist die Lernkurve gezeigt, auf der jeweils die Veränderung des Fehlers jeweils für den Trainings- und Validierungsdatensatz zu sehen ist. Der Graph zeigt, dass das Modell bereits ab c.a. Epoche 20 keine signifikante Verbesserung auf dem Validierungsdatensatz erreicht. Lediglich aufgrund des Early-Stoppings wurde das Modell noch weiter trainiert, bis endgültig keine Verbesserung nach mindestens $patience = 10$ Trainingszyklen vorhanden war. Letztendlich beträgt der Fehler auf den Validierungsdaten $mae_{simple\_dense} \approx 59828492$, was um c.a. 21 Millionen bzw. c.a. 23.7% besser als der Fehler der Mittelwertsprognose ist. Sowohl der

Code, als auch einige Hyperparameter (wie Aktivierungsfunktion und Anzahl von Neuronen) sind im Minimalbeispiel 2 im Anhang zu finden.

Einschub Convolution

Im Bereich der Computer Vision ist es üblich, Filter auf Bilder anzuwenden, um gewisse Eigenschaften zu extrahieren. So sind beispielsweise sowohl der Laplace als auch der Sobel Filter bekannte Filter, welche es ermöglichen Kanten aus Bildern hervorzuheben. [Kan88]
Im Bereich der neuronalen Netze werden sog. Convolution Schichten benutzt, um Filter auf die Daten anzuwenden. Dabei haben diese Filter jedoch, im Gegensatz zu den o.g. klassischen Verfahren, trainierbare Gewichte. Deshalb ist es ihnen möglich, sich an die Daten und das Problem anzupassen, um somit mehrere verschiedene Filter zu lernen, die dabei jeweils andere Merkmale aus den Bildern extrahieren. [O'S15]

3.2 Vorhersage mit einem Convolution Filter

Abb. 13 zeigt den bekannten Eingabetensor mit drei vergangenen Tagen und jeweils drei Attributen. Der sog. $kernel\_size = 3$ Parameter der verwendeten Convolution Schicht gibt die „Breite" des Filters an und bezieht in diesem konkreten Fall Gewichte zu allen Attributen aller Tage, um genau eine Ausgabe zu produzieren. Dabei gibt der Parameter $filters = 4$ die Anzahl der Filter einer einzigen Schicht an. Das besondere dabei ist, dass jeder Filter einen separaten Satz an Gewichten hält und somit hier vier verschiedene Filter insgesamt 4 Ausgaben produzieren.

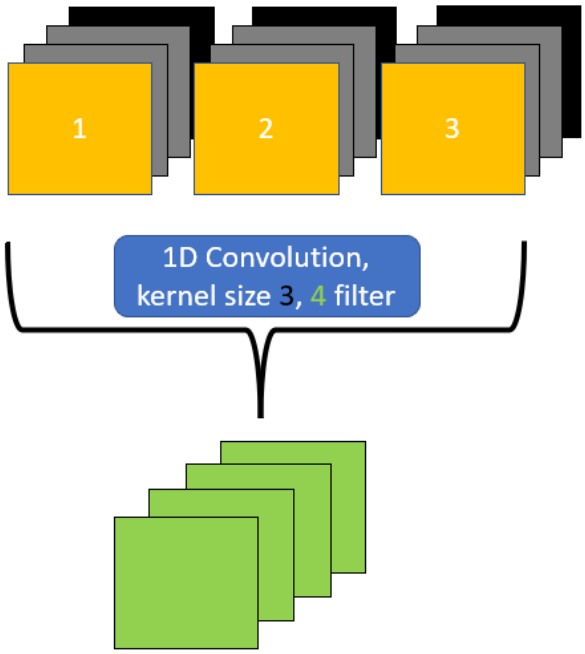


Abbildung 13: Convolution extrahiert die wichtigsten Merkmale des Eingangstensors. Der Batch wurde absichtlich ausgelassen. Quelle: Eigene Darstellung

So hat wieder der Eingabetensor das Format $shape = (32, 3, 3)$ und der Tensor, der nach der Convolution Schicht entsteht, $shape = (32, 1, 4)$. Abschließend muss dieser, analog zum ersten Schritt in Abb. 11, in das Format $shape = (32, 4)$ (der überschüssige Rang wird

entfernt) gebracht werden, einer Ausgabeschicht übergeben und abschließend in das Format $shape = (32, 1 * 1)$ gebracht werden.

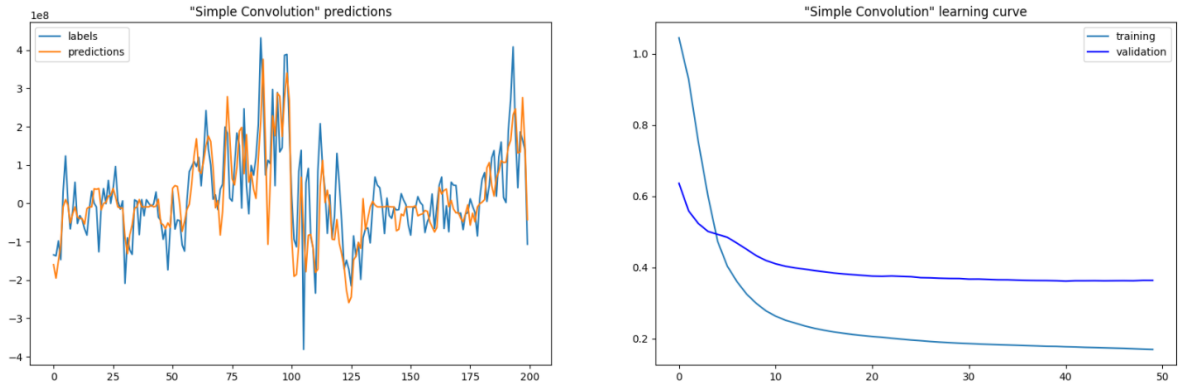


Abbildung 14: Nahezu identisches Lernverhalten als auch Vorhersage wie beim einfache Dense Modell. $mae_{simple\_conv} \approx 59634976$. Quelle: Eigene Darstellung

Abb. 14 zeigt, dass sowohl das Lernverhalten, als auch die Vorhersagekurve große Ähnlichkeit zum vorher gezeigten Modell mit einer verborgenen Schicht hat. Bei der Evaluation stellt sich heraus, dass der Fehler bei $mae_{simple\_conv} \approx 59634976$ liegt und somit nur marginal besser als beim vorherigen Modell ist.

Einschub rekurrente neuronale Netze

Rekurrente neuronale Netze ähneln den zuvor besprochenen Netzen insofern, dass diese ebenfalls Zellen besitzen, Gewichte zur Eingabe haben, eine Eingabe entgegennehmen und letztendlich eine Ausgabe produzieren. Allerdings kann eine rekurrente neuronale Schicht als zeitgesteuerte Funktion gesehen werden, da die Ausgaben von den vorherigen Eingaben abhängen. [Gé19]

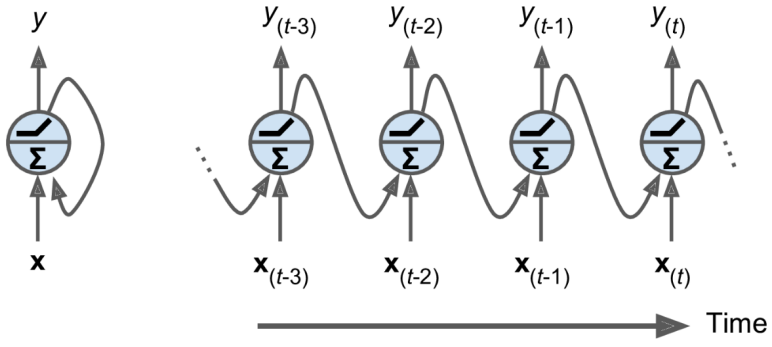


Abbildung 15: RNN Zelle aufgerollt über die Zeit. Quelle: [Gé19]

Abb. 15 zeigt auf der linken Seite eine Recurrent Neural Network (RNN) Zelle, welche über die Zeit aufgerollt ist. Wie zu sehen ist, ist die nächste Ausgabe der Zelle abhängig von der letzten Eingabe, da die RNN Zelle intern einen Zustand akkumuliert, welcher als Eingabe für die nächste Ausführung der Zelle zwischengespeichert wird. Anders als die klassische Zelle hat die RNN Zelle separate Gewichte für die Eingabe und den akkumulierten Zustand. Sei W_x die Matrix, welche die Gewichte für die Eingaben hält, W_y die Matrix, welche die Gewichte für

die Ausgabe des Zeitschritts $t-1$, X_t die Daten für den aktuellen Zeitschritt t und Y_{t-1} die Ausgabe vom vorherigen Zeitschritt $t-1$. Dann zeigt die Gleichung

$$Y_{(t)} = \phi(X_{(t)}W_x + Y_{(t-1)}W_y + b) \tag{3}$$

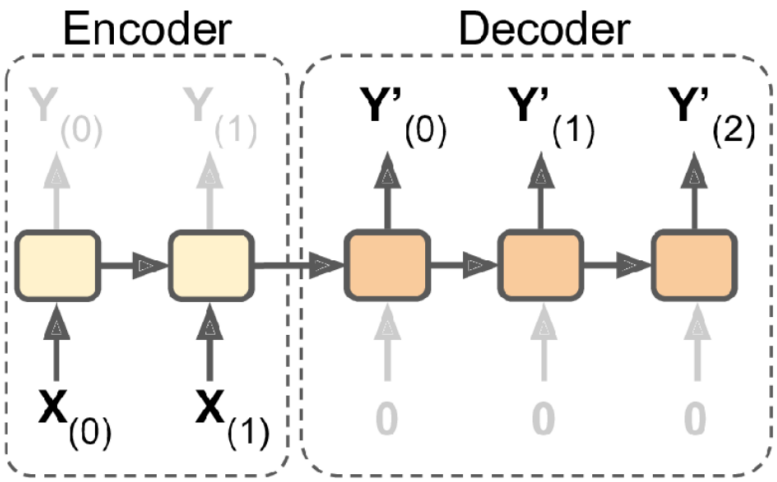
wie die Ausgabe einer RNN Zelle berechnet wird. Somit ist der Unterschied zur klassischen neuronalen Zelle, die mit $\phi(XW + b)$ definiert ist, die gewichtete Summe der vorherigen Ausgabe. Jedoch kann der akkumulierte Zustand, welcher als Ausgabe des letzten Zeitschritts beschrieben wurde, auch anderes berechnet werden. Dies ist beispielsweise bei komplexeren Varianten, wie z.B. der LSTM Zelle (die gleich aufgegriffen wird), der Fall.
Mithilfe der RNN Zellen lässt sich die sog. „Encoder-Decoder“ Architektur realisieren. Diese findet z.B. im Sprachverarbeitungsbereich seine Verwendung, da hier ebenfalls Sequenzen zu neuen Sequenzen verarbeitet werden (z.B. Übersetzung). Die Idee bei der Architektur (s. Abb. 16a) ist, eine Sequenz mit einem Encoder zu enkodieren und anschließend mit einem Decoder zu der gewünschten Sequenz zu dekodieren. [Sut14]

Abb. 16b zeigt im Sinne der vorherigen Beispiele, wie jeder Tag nacheinander in eine RNN Zelle eingespeist wird. Der daraus resultierende verborgene Zustand wird abschließend einer neuen rekurrenten Zelle als Startzustand gegeben. Diese nimmt den Zustand entgegen, setzt ihn als eigenen verborgenen Zustand und bekommt den ersten Tag als Eingabe. Zuletzt wird im rekursiven Stil die Ausgabe der Zelle als nächste Eingabe verwendet, bis die gewünschte Anzahl an vorherzusagenden Tagen generiert worden ist. Dabei repräsentieren die grünen Blätter in Abb. 16b die endgültige Ausgabe des Netzes.

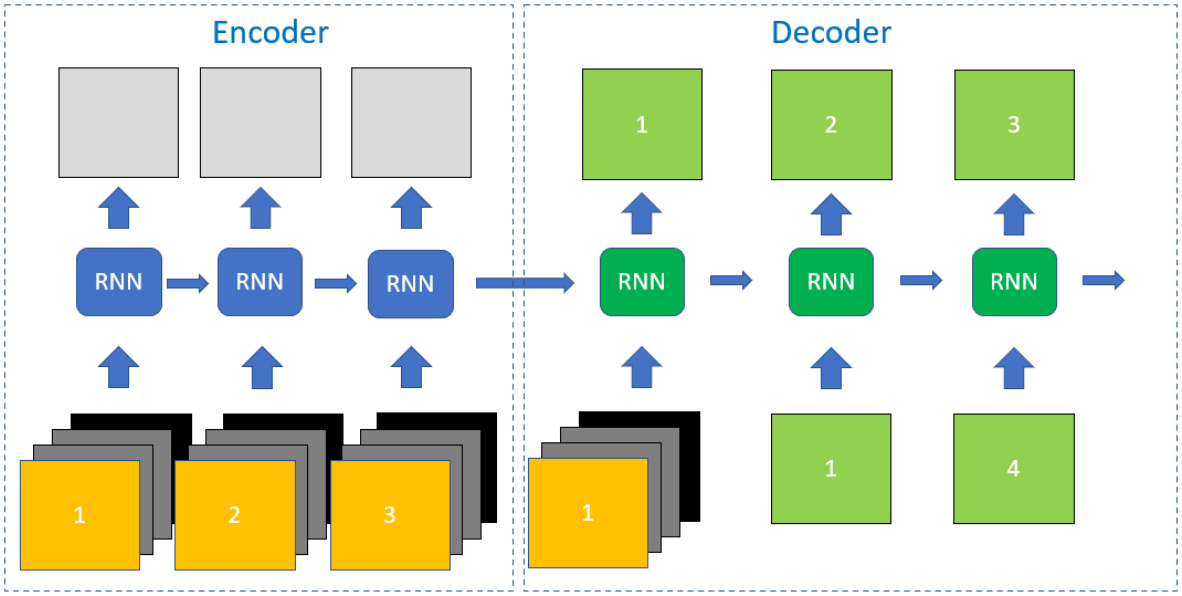
3.3 Vorhersage mit LSTM Zellen

Die klassische RNN Zelle hat jedoch den Nachteil, dass lange Sequenzen nicht effektiv enkodiert werden. Deshalb gibt es verbesserte Varianten, wie z.B. die RNN oder Gated Recurrent Unit (GRU) Zelle. Diese verfügt über eine komplexere Architektur, wie z.B. eine „Forget Gate“, mithilfe derer die RNN Zelle redundante Eingaben vergessen kann. [Hoc97]
Sie wird hier nicht näher erläutert, da dies den Rahmen dieser Arbeit sprengen würden. Im folgenden werden jeweils statt den herkömmlichen RNN Zellen jeweils LSTM Zellen verwendet. Für die folgenden Tests wurde für die bisherige 1-Tages-Vorhersage nur der Encoder Teil der in Abb. 16b gezeigten Architektur verwendet. Die letzte Ausgabe der RNN Zelle wurde somit an eine klassische, voll verbundenen Schicht weitergeleitet, welche in dem konkreten Fall nur aus einer Zelle besteht (da nur ein Tag mit einer Zielvariable vorhergesagt werden soll) und demnach der Ausgabeschicht entspricht. Das Format des Eingabetensors ist $shape = (32, 3, 3)$, die RNN Zelle bekommt jeweils einen Tag aus dem zweiten Index und es wird letztendlich nur die letzte Ausgabe wiederverwendet, welche das Format $shape = (32, 4)$ hat (vier ist die Anzahl der RNN Zellen in einer Schicht). Die Ausgabeschicht produziert daher einen Tensor mit Format $shape = (32, 1)$ und dies kann durch Hinzunahme eines redundanten Rangs auf $shape = (32, 1, 1)$ erweitert und somit in das Format der Labels gebracht werden.

Abb. 17 zeigt die Lernkurve als auch die Vorhersage auf dem Validierungsdatensatz für das LSTM Modell. Hierbei ist zu sehen, dass die Vorhersage für einige Ausreißer deutlich besser ausfällt, als bei den vorherigen Modellen. So weicht das Convolution Modell (in Abb. 14 auf der linken Seite zu sehen) zwischen Tag 75 und Tag 100 deutlich stärker vom Zielwert aus, als das LSTM Modell. Der Fehler von c.a. $mae_{simple\_lstm} \approx 56015776$ bestätigt, dass das LSTM Modell besser abschneidet.
Da im Wettbewerb auch eine Langzeitvorhersage gefordert ist (Tag 2 bis Tag 5) wurde die in Abb. 16b beschriebene Architektur genutzt, um vier Tage vorherzusagen. Ebenso wurde



(a) Klassischer Encoder-Decoder Stack. Quelle: [Gé19]



(b) Variation der klassischen Encoder-Decoder Architektur für gegebenes Zeitreihenproblem. Quelle: Eigene Darstellung

ein Modell für eine 5-Tages-Vorhersage mithilfe der in Kap. 2.4 vorgestellten gewichteten Verlustfunktion implementiert. Da die Resultate jedoch nicht im Vergleich zu der 1-Tages-Vorhersage stehen, sind sie hier nicht weiter aufgeführt.

3.4 Kombination verschiedener Daten und komplexere Architekturen

Wie bereits in Kap. 1.5 besprochen, haben die Tensoren der Eingabedaten ein fest definiertes Format. So ist beispielsweise in den letzten Beispielen das Format $shape = (32, 3, 3)$. Da im Rahmen des Wettbewerbs jedoch auch zukünftige Wetterdaten bereitstehen (für die jeweils zu vorhersagenden Tage), wurden diese ebenfalls mit den in Kap. 1.5 vorgestellten Methoden in Tensoren verarbeitet. Dadurch entsteht eine zunehmende Komplexität des Modells, da das Modell mehrere Eingaben hat.

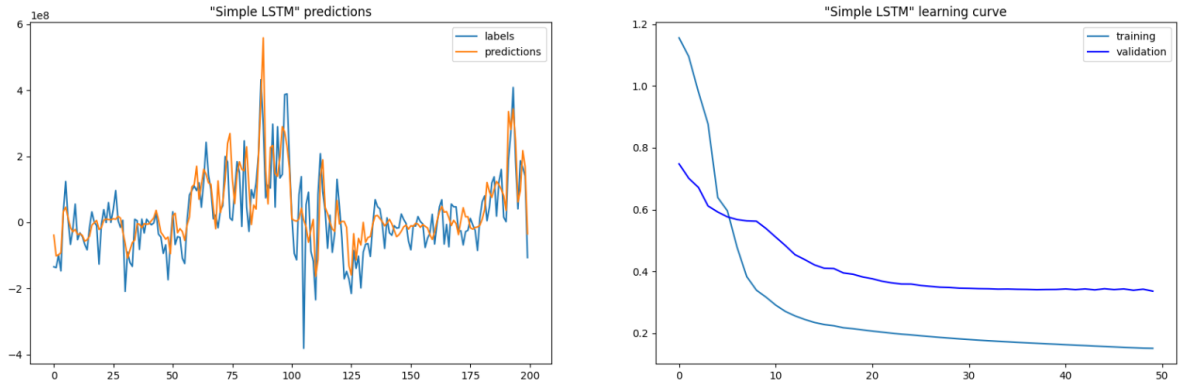


Abbildung 17: 1-Tages-Vorhersage schneidet mit LSTM Zelle besser ab. $mae_{simple\_lstm} \approx 56015776$. Quelle: Eigene Darstellung

Insbesondere kann nicht jeder beliebige Tensor mit einem jeweils anderem beliebigen Tensor konkateniert oder deren Inhalte addiert werden. In diesem Fall beträgt das Format des Tensors mit den zukünftigen Wetterdaten $shape = (32, 1, f)$. Dabei hat der mittlere Index die Dimension eins, da nur ein Tag vorhergesagt wird und f ist die Anzahl der exogenen Wetterattribute. Um also eine Addition mit den Inhalten beider Tensoren durchzuführen, müssen die beiden letzten Indizes gleich viele Dimensionen haben. Im Fall der Konkatenation reicht es, wenn der letzte Index beider Tensoren gleich viele Dimensionen hat. [Dul10]
Abb. 18 zeigt, wie der Tensor mit Wetterdaten mithilfe einer voll verbundenen Schicht in ein für die Konkatenation mit dem Eingabetensor kompatibles Format gebracht werden kann.

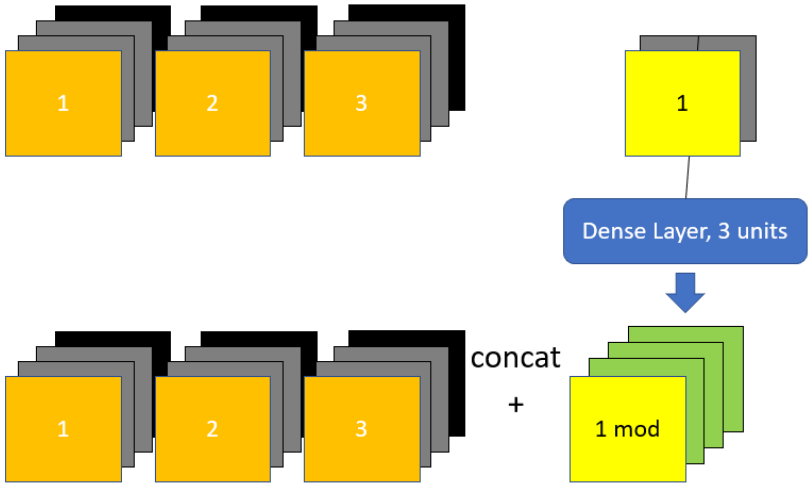


Abbildung 18: Vorbereitung für Konkatenation verschiedenartiger Tensoren. Quelle: Eigene Darstellung

Bei den Tests im Rahmen der Arbeit hat sich jedoch ergeben, dass die Verwertung der zusätzlichen Daten nicht in allen Fällen das Ergebnis verbessert. Hier ist denkbar, dass es noch bessere Ansätze für die Kombination der Daten, abgesehen von den bereits vorgestellten, gibt. Die alternative Interpretation ist, dass die Zukunftsdaten die Vorhersage erschweren, da sie nicht zur Lösung des Problems beitragen und somit das neuronale Netz mehr unnötige Informationen zu verwerten hat.
Letztendlich wurde auch eine andere, komplexere Architektur implementiert und getestet.

Hierbei handelt es sich um eine Mischung von allem Convolution, LSTM und des Multi-Headed-Attention Mechanismus (welcher aus dem Sprachverarbeitungsbereich stammt und hier nicht näher erklärt wird, siehe [Vas17]). Laut den Autoren der Architektur glätten die Convolution Schichten die Variablen und reduzieren somit die Varianz (dabei wird eine Filter-Breite von zwei verwendet). Die Multi-Headed-Attention Schicht versucht die wichtigsten Resultate der Convolution Schicht zu extrahieren und bringt diese ebenso in einen niedriger dimensionalen Raum. Letztendlich wird mithilfe von LSTM-Schichten die zeitliche Reihenfolge der Daten kodiert und letztendlich eine Vorhersage getätigt. [Bu20]
Beim Test fiel jedoch auf, dass die Fehler im Durchschnitt um 30% schlechter als die bei den in Kap. 3.1, 3.2 und 3.3 vorgestellten Architekturen waren. Auch die Reduzierung der Anzahl von Neuronen in den jeweiligen Schichten konnte die Architektur nicht auf die Fehlerrate der einfachen Architekturen bringen. Hier ist denkbar, dass ein großes neuronales Netz mit vielen Schichten und Neuronen um ein vielfaches mehr an Trainingsdaten braucht, als es in dieser Arbeit bzw. dieses Wettbewerbs der Fall ist.

4 Zusammenfassung

4.1 Optimierung mittels Gittersuche

Um ein möglichst optimales Ergebnis zu erreichen, wurden die in den vorherigen Kapiteln erklärten einfachen Architekturen mithilfe einer Gittersuche zur Probe gestellt. Dabei wurde auf Basis des bereits vorgestellten Test- bzw. Validierungsdatensatzes jede Kombination von einer fest definierten Menge an Hyperparametern getestet. Dabei wurden folgende Hyperparameter variiert:

- Aktivierungsfunktion: SELU, ReLU, Swish
- Anzahl an Neuronen: 8, 16, 32, 64, 128
- Batchnormalisierung: An, Aus
- Dropout: 0.0, 0.2, 0.4
- Zukunftsdaten: An, Aus
- lag p: 1,2,6,7
- saisonaler lag m: 6,7,14

Dabei ist *Dropout* ein Mechanismus, welcher beim Training einen prozentualen Anteil der Neuronen aussetzt. Dadurch passt sich das Modell schwächer an die Trainingsdaten an (es „overfittet“ nicht) und generalisiert besser. Die Batchnormalisierung kann vor oder nach einer Aktivierungsfunktion eingesetzt werden. In diesem Fall wird sie nach jeder Aktivierungsfunktion eingesetzt, und normalisiert die Ausgänge der Aktivierungsfunktionen auf Standarabweichung $\sigma \approx 1$ und Mittelwert $\mu \approx 0$.

Das NNAR ist dabei ein Modell mit nur einer verborgenen Schicht. Der Unterschied zu dem in Kap. 3.2 gezeigten Modell ist, dass Zukunftsdaten mit dem in Kap. 3.4 vorgestellten Verfahren integriert werden.
In Tab. 2 und 3 sind die Ergebnisse zu sehen. Dabei ist deutlich zu erkennen, dass die Batchnormalisierung in jedem Fall schlechter ist, was daran liegen kann, dass die Daten bereits normalisiert sind. Andersherum ist Dropout ein wichtiger Parameter, da somit das Modell besser auf den Testdaten generalisiert. Ebenso ist zu sehen, dass der Dropout Wert mit der

Tabelle 2: Bestenliste aufsteigend

Name	**mae_one_day**	**norm**	**dropout**	**n**	**activation**	**p**	**m**
NNAR 0	59546950.0	False	0.2	16	swish	1	6
NNAR 1	59719172.0	False	0.4	128	relu	6	7
NNAR 2	60251570.0	False	0.4	128	swish	6	14
NNAR 3	60274456.0	False	0.2	16	relu	1	7
NNAR 4	60284184.0	False	0.4	128	relu	1	7

Tabelle 3: Bestenliste absteigend

Name	**mae_one_day**	**norm**	**dropout**	**n**	**activation**	**p**	**m**
NNAR n-4	85122550.0	True	0	64	relu	6	7
NNAR n-3	86228110.0	True	0	64	swish	6	7
NNAR n-2	86591944.0	True	0	128	relu	6	7
NNAR n-1	86897656.0	True	0.2	8	swish	6	14
NNAR n-0	93530920.0	True	0	8	swish	6	7

Anzahl der Neuronen korreliert. Dies bedeutet, dass das Modell einen höheren Anteil an Neuronen beim Training ignorieren muss, wenn die Anzahl der Neuronen im Modell hoch ist. Letztendlich erweist sich die Kombination mit der Swish Aktivierungsfunktion, $p = 1$ und $m = 6$ als beste Parameter. Dies bestätigt die Vermutung, dass der vorherige Tag und der sechste Tag vor der Prognose stark mit dem zu vorhersagenden Tag korrelieren. Dies ist auch in Kap. 1.3 und Abb. 2 zu sehen und bestätigt die erste Vermutung.
Es ist wichtig zu erwähnen, dass diese Tests mit einer Sonderbedingung des Wettbewerbs durchgeführt worden sind. Dabei wird der erste Tag innerhalb der 6-Tages-Lücken nicht gewertet, sondern muss übersprungen werden. Dies erschwert die Vorhersage deutlich, weshalb hier ein Fehler von $mae_{nnar_0} \approx 59546950$ am besten ist. Die in den vorherigen Kapiteln vorgestellten Modelle haben schlechter abgeschnitten, weshalb sie hier ausgelassen worden sind.

4.2 Fazit

Insgesamt kann gesagt werden, dass die Intuition hinter den neuronalen Netzen zur Zeitreihenvorhersage schwer zu erklären ist. So scheint die LSTM Zelle als wäre sie perfekt für eine Zeitreihenvorhersage und schneidet in Kap. 3 am besten ab. Sobald jedoch die Problemstellung mit der im letzten Abschnitt genannten Sonderbedingung geändert wird, ist das NNAR und die Integration von zukünftigen Daten besser. Auch die komplexeren Architekturen konnten in diesem Test nicht überzeugen. Dies kann offenbar daran liegen, dass eine solche Architektur ebenso an ein gewisses Problem bzw. an eine gewisse Menge an Trainingsdaten gebunden ist. Letztendlich braucht das gewählte Modell eine Gittersuche, um die optimalen Parameter zu finden. Hierbei ist zu sehen, dass die anfänglichen Vermutungen über die Korrelation vergangener Tage übereinstimmen.
Es ist schlussendlich anzumerken, dass die neuronalen Netze brauchbare Ergebnisse bei Zeitreihenproblemen liefern. So übertrifft das zuletzt vorgestellte Modell die Ergebnisse klassischer Verfahren, wie XGB oder SARIMA (auf welche hier nicht näher eingegangen wurde, jedoch trotzdem eine Evaluation geschehen ist). Dies ist definitiv der Optimierung mittels Gittersuche zu danken, was zeigt, dass das Thema Hyperparametertuning eine große Rolle bei neuronalen

Netzen spielt.

A Anhang

Aus Gründen der Übersicht wurde nur jeweils eins der verschiedenen Modelle als Codebeispiel angehängt. Die Datenvorverarbeitungsroutinen wurden ebenfalls ausgelassen.

A.1 Verlustfunktionen

Listing 1: Verlustfunktion

```
class WeightedLoss:
def __init__(self, weights) -> None:
    self.weights = tf.convert_to_tensor(weights)

def __call__(self, y_true, y_pred):
    y_pred = tf.convert_to_tensor(y_pred)
    y_true = tf.cast(y_true, y_pred.dtype)

    squared_loss = tf.math.abs(tf.math.subtract(y_pred, y_true))
    reduced = tf.reduce_mean(input_tensor=squared_loss, axis=-1) # or squeeze

    calculated_shape = tf.shape(reduced)
    broadcasted_weights = tf.broadcast_to(self.weights, calculated_shape)

    # Mit Hilfe von sample_weight wird jeder Datenpunkt
    # (z.B. 3 Tage) mit einem custom weight "gemittelt"
    # Zum schluss muss der Durchschnitt vom Batch berechnet werden.
    reduced = tf.multiply(reduced, broadcasted_weights)
    reduced = tf.reduce_sum(reduced)
    reduced = tf.divide(reduced, tf.cast(calculated_shape[0], y_pred.dtype))

    return reduced
```

A.2 Modelle

Listing 2: Einfache verborgene Schicht

```
simple_dense = tf.keras.Sequential(
    [
        # lag = 3
        tf.keras.layers.Lambda(lambda x: x[:,-3:,:]),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(4, activation="relu"),
        tf.keras.layers.Dense(1, kernel_initializer="uniform"),
        tf.keras.layers.Reshape((1,1))
    ]
)
```

Listing 3: NNAR Modell

```
class NNAR(BaseModel):
```

```
def __init__(self, activation="selu", m=1, p=1, **kwargs):
    super().__init__(dropout_count=1, batch_norm_count=1, **kwargs)

    self.m = m
    self.p = p

    ### INIT LAYERS...


def call(self, inputs, training=None):
    # x = past data
    x = inputs["a"]
    # b = future data (less features than x)
    b = inputs["b"]

    lags = self.flatten(x[:,-self.p:,:])

    if self.m > self.p:
        s_lags = self.flatten(x[:,-self.m,:])
        lags = tf.concat([s_lags,lags], axis=1)

    # combine
    x = lags

    b = self.flatten(b)
    x = tf.concat([x,b], axis=1)

    # hidden layer
    x = self.dense_1(x)
    if self.batch_norm:
        x = self.batch_norm_1(x)
    if self.has_dropout:
        x = self.dropout_1(x, training=training)

    x = self.dense_out(x)

    # reshape to match output
    x = tf.reshape(x, [tf.shape(x)[0], self.out_steps, self.out_features])

    return x
```

Listing 4: NNAR Modell

```
class DenseLstmDense(BaseModel):
def __init__(self, cell=tf.keras.layers.LSTMCell, m=0, p=1, bi_encoder=False, ac
    super().__init__(batch_norm_count=2, dropout_count=2, **kwargs)

    self.p=p
    self.m=m
```

```
    ### INIT LAYERS...


def call(self, inputs, training=None):
    # x = past data
    x = inputs["a"]
    # b = future data (less features than x)
    b = inputs["b"]

    # LAGS
    lags = x[:,-self.p:,:]
    if self.m > self.p:
        lag_day = tf.expand_dims(x[:,-self.m,:], axis=1)
        lags = tf.concat([lag_day,lags], axis=1)

    x = lags
    # DENSE
    x = self.dense_1(x)
    if self.batch_norm:
        x = self.batch_norm_1(x, training)
    if self.has_dropout:
        x = self.dropout_1(x, training)
    b = self.dense_b(b)
    if self.batch_norm:
        b = self.batch_norm_1(b, training)
    if self.has_dropout:
        b = self.dropout_1(b, training)

    x = tf.concat([x,b], axis=1)

    # ENCODER
    x = self.encoder(x)

    # DENSE
    x = self.dense_2(x)

    # OUTPUT
    x = self.dense_out(x)

    # reshape to match output
    x = tf.reshape(x, [tf.shape(x)[0], self.out_steps, self.out_features])

    return x
```

Literatur

[Bu20] S.-J. Bu und S.-B. Cho. Time Series Forecasting with Multi-Headed Attention-Based Deep Learning for Residential Energy Consumption. *Energies*, 13(18), 2020.

[Dul10] K. Dullemond und K. Peeters. Introduction to Tensor Calculus, 2010.

[Gé19] A. Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent.* O'Reilly Media, 2nd ed. Aufl., 2019.

[Hoc97] S. Hochreiter und J. Schmidhuber. Long Short-term Memory. *Neural computation*, 9:1735–80, 12 1997.

[Kan88] N. Kanopoulos, N. Vasanthavada und R. Baker. Design of an image edge detection filter using the Sobel operator. *IEEE Journal of Solid-State Circuits*, 23(2):358–367, 1988.

[Kin14] D. P. Kingma und J. Ba. Adam: A Method for Stochastic Optimization, 2014.

[Kla17] G. Klambauer, T. Unterthiner, A. Mayr und S. Hochreiter. Self-Normalizing Neural Networks. *CoRR*, abs/1706.02515, 2017.

[Nwa18] C. Nwankpa, W. Ijomah, A. Gachagan und S. Marshall. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *CoRR*, abs/1811.03378, 2018.

[Ora17] F. Orabona und T. Tommasi. Backprop without Learning Rates Through Coin Betting. *CoRR*, abs/1705.07795, 2017.

[O'S15] K. O'Shea und R. Nash. An Introduction to Convolutional Neural Networks. *CoRR*, abs/1511.08458, 2015.

[Ram17] P. Ramachandran, B. Zoph und Q. Le. Swish: a Self-Gated Activation Function. 10 2017.

[Rud16] S. Ruder. An overview of gradient descent optimization algorithms, 2016.

[Sch01] R. Schlittgen und B. Streitberg. *Zeitreihenanalyse.* Lehr- und Handbücher der Statistik. De Gruyter, 2001.

[sta22] statsmodels. Introduction - statsmodels. `https://www.statsmodels.org`, 2022.

[SUT86] R. SUTTON. Two problems with back propagation and other steepest descent learning procedures for networks. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society, 1986*, S. 823–832, 1986.

[Sut14] I. Sutskever, O. Vinyals und Q. V. Le. Sequence to Sequence Learning with Neural Networks. *CoRR*, abs/1409.3215, 2014.

[Vas17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser und I. Polosukhin. Attention Is All You Need, 2017.

[Xu20] J. Xu, Z. Li, B. Du, M. Zhang und J. Liu. Reluplex made more practical: Leaky ReLU. S. 1–7. 07 2020.