Gymnasium

Straße 1

12345 Stadt

Facharbeit im Seminarfach

Künstliche neuronale Netze

Wie trifft ein künstliches neuronales Netz eine Voraussage?

Verfasser: **schneefux**

Fachlehrer: **xxx**

Abgabedatum: **15. März 2016**

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Wie trifft ein künstliches neuronales Netz eine Voraussage?
Ein künstliches neuronales Netz ist durch immer schneller werdende Computer die de facto Lösungsmethodik, um von einem Problem zu einer Lösung zu kommen, ohne den direkten Weg dorthin zu kennen. Mit riesigen Datenmengen können Firmen wie Google und Facebook Computergehirne trainieren, um anschließend das Verhalten ihrer Nutzer vorherzusagen. Der Traum eines jeden Menschen ist es, das Unbekannte und die Zukunft zu erfahren. Künstliche neuronale Netze bringen die Technologie ein Stück weiter in die Richtung, in der Regressionen und strukturelle Algorithmik an ihre Grenzen gelangt sind.
In dieser Facharbeit soll es um die Funktionsweise eines kleinen Feedforward-Netzes gehen, welches darauf trainiert wird, anhand verschiedener Kriterien eine Marktpreiseinschätzung zu geben. Feedforward-Netze sind die einfachsten Typen künstlicher neuronaler Netze, die sich für eine Vorhersage von Zahlenwerten eignen. Das künstliche neuronale Netz wird mit Python und dem Tensorflow-Framework implementiert. Dahinführend, wird zunächst die biologische Grundlage der natürlichen neuronalen Netze aufgezeigt. Anschließend wird von dieser Grundlage ausgegangen, und die wesentlichen Funktionselemente eines biologischen Neurons werden auf ein künstliches Neuron übertragen, welches durch einfache mathematische Operationen im Verbund mit anderen zum Lösen von Problemen verwendet werden kann. Schließlich wird an der selbst geschriebenen Beispielimplementierung eines künstlichen neuronalen Netzes demonstriert, wie dieses dem Netz unbekannte Hauspreismittelwerte in der Region Bostons anhand einer Reihe von Umgebungsfaktoren auf bis zu 500$ genau bestimmt. Anhand dieses Programmes werden die Auswirkungen der Veränderung verschiedener Konfigurationsparameter des künstlichen neuronalen Netzes untersucht, um zu verstehen, wie sie die Vorhersagequalität beeinflussen. Als Abschluss wird auf die weiteren Möglichkeiten künstlicher neuronaler Netze eingegangen.

1.1 Biologische Grundlage

Neuronen (Abbildung 1) bestehen aus Dendriten und Axonen. Ein Signal wird über die Dendriten aufgenommen und bei Überschreiten des Schwellwerts des sogenannten Axonhügels am Axon als elektrisches Signal zur nächsten Zelle weitergeleitet. Die Verbindung zur nächsten Zelle wird Synapse genannt. Neuronale Netze bestehen aus einer großen Menge an Neuronen und Synapsen. Das menschliche Gehirn besitzt rund 100 Milliarden Neuronen, von denen jedes einzelne mit durchschnittlich 1000

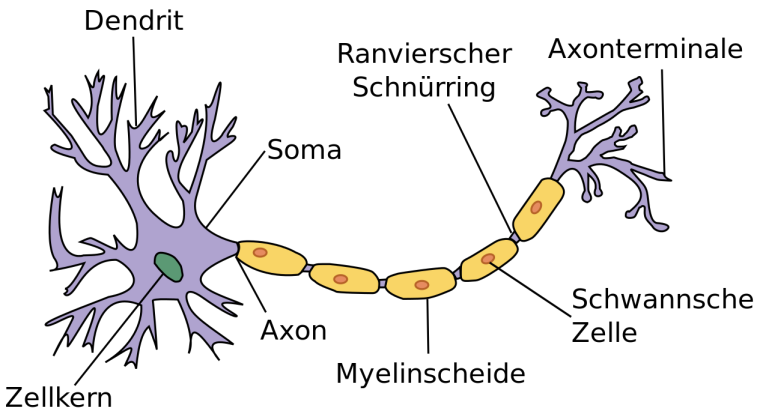

Abbildung 1: Schema eines Neurons[2]

anderen verbunden ist[1].

Die von den Dendriten aufgenommenen Eingangssignale können Ausgangssignale eines anderen Neurons sein oder Signale eines Rezeptors. Rezeptoren sind zum Beispiel Photorezeptoren im Auge oder Dehnungsrezeptoren an Sehnen[3]. Geht ein Ausgangssignal nicht zu einem weiteren Neuron, geht es zu einem Effektor. Ein Effektor kann eine Muskelzelle sein[4]. Die Funktion eines Neurons lässt sich am Patellarsehnenreflex beobachten. Unterhalb der Kniescheibe sitzen Dehnungsrezeptoren, die einen Schlag registrieren und ein Signal zum Rückenmark senden, von wo aus es direkt über eine einzelne Synapse an den Effektor Muskel geht und eine Streckung des Beins verursacht[5]. Wenn des Eingangssignal zu schwach ist, um den Axonhügel zu passieren, wird das Bein nicht gestreckt.

In einem neuronalen Netz sind die Informationen durch die Eigenschaft der Synapsenverbindungen, bei jeder Nutzung stärker zu werden, in diesen gespeichert. Ein biologisches Gehirn lernt von alleine durch Wiederholungen.

1.2 Entwicklung künstlicher neuronaler Netze

Zwischen 1942 und 1955 gab es die ersten Gedanken zu künstlichen neuronalen Netzen, 1943 von Walter Pitts und Warren Mc Culloch eine theorethische Umsetzung, allerdings ohne Anwendungsbezug. Bis etwa 1969 wurden viele Grundbausteine der Forschung gelegt. Das erste tatsächlich funktionsfähige künstliche neuronale

[1] Vgl. Lossau, Norbert: Die Welt. In: Forscher wollen die Highways im Gehirn nachbauen, http://www.welt.de/wissenschaft/article124674193/Forscher-wollen-die-Highways-im-Gehirn-nachbauen.html, 09. 02. 2014, Stand: 13. März 2016.

[2] SEER, US National Cancer Institute, Dhp1080 und Unbekannt, Wikimedia Commons: Neuron, https://commons.wikimedia.org/wiki/File:Neuron_(deutsch)-1.svg, 19. 12. 2006, Stand: 12. März 2016

[3] Vgl. Antwerpes, Frank: DocCheck Flexikon. Das Medizinlexikon zum Medmachen. In: Rezeptor, http://flexikon.doccheck.com/de/Rezeptor, 19. 01. 2015, Stand: 12. März 2016.

[4] Vgl. Könneker, Carsten und Reichert, Uwe: Lexikon der Biologie. In: Effektorzellen, http://www.spektrum.de/lexikon/biologie/effektorzellen/20145, 1999, Stand: 12. März 2016.

[5] Vgl. Weber, Michael: Webmic's Page. In: Reflexe, http://www.webmic.de/reflexe.htm, 2000, Stand: 12. März 2016.

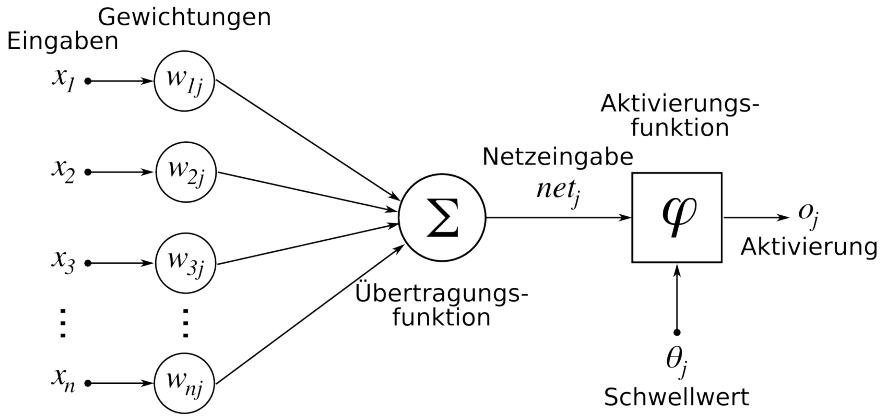

Abbildung 2: Schema eines künstlichen Neurons[7]

Netz wurde 1957/1958 von Frank Rosenblatt und Charles Wightman entwickelt, das Mark I Perceptron war eine Ziffernerkennung gebaut mit motorbetriebenen Potentiometern. Weitere Forschung verlief bis 1985 schleichend. 1974 entwickelte Paul Werbos das Fehlerrückführungsverfahren. Durch die Innovation der Hopfield-Netze, die in der Lage waren, schwierige Optimierungsaufgaben zu lösen, kam das Thema wieder in den Fokus vieler Forscher und sorgt bis heute für Innovationen.[6]

Künstliche neuronale Netze gehören zur Klasse der neuronalen Netze, ebenso wie ein biologisches neuronales Netz. Ein Gehirn ist ein besonders großes biologisches neuronales Netz. Künstliche neuronale Netze haben nicht das Primärziel, die Funktionsweise eines biologischen neuronalen Netzes exakt nachzubilden, sind also nicht Untersuchungsgegenstand der Hirnforschung. Stattdessen haben künstliche neuronale Netze biologische als Vorbild. Ein künstliches Neuron simuliert die wesentlichen Elemente eines biologischen Neurons.

1.3 Funktionsweise künstlicher neuronaler Netze

Im folgendem Abschnitt wird die Funktionsweise eines biologischen Neurons auf ein künstliches Neuron abstrahiert.

Ein Schema eines einzelnen Neurons ist in Abbildung 2 dargestellt. Die Dendriten werden durch Eingaben $x_1, x_2, \ldots x_n$ dargestellt. Ein biologisches Neuron hat stärkere und schwächere Synapsen, deren Stärke die Priorität eines Signals beeinflusst. In einem künstlichen Neuron wird das durch die Gewichte $w_1, w_2, \ldots w_n$ abgebildet. Die Werte der Eingaben werden mit den Gewichten multipliziert und anschließend summiert. Diese gewichtete Summe wird Netzeingabe gennannt und berechnet sich als

[6] Vgl. Kriesel, David: Ein kleiner Überblick über Neuronale Netze. In: Zur Geschichte Neuronaler Netze, S. 10ff. http://www.dkriesel.com/_media/science/neuronalenetze-de-zeta2-1col-dkrieselcom.pdf, 2007, Stand: 12. März 2016.

[7] Chrislb, Wikimedia Commons: Schema eines künstlichen Neurons, https://commons.wikimedia.org/wiki/File:ArtificialNeuronModel_deutsch.png, 14. 07. 2005, Stand: 12. März 2016

$net = \sum_n (x_n \cdot w_n)$. In einem biologischen Neuron entscheidet der Schwellwert über die Weitergabe eines elektrischen Signals. Bei künstlichen neuronalen Netzen existieren verschiedene Varianten, die sich als Aktivierungsfunktion zusammenfassen lassen. Es ist möglich, dass die Aktivierungsfunktion nur dann eine Ausgabe liefert, wenn ein Schwellenwert überschritten wird – dann ist es eine Schrittfunktion, meistens wird aber als Aktivierungsfunktion eine lineare Funktion, eine Sigmoidfunktion oder eine Softmaxfunktion genutzt.[8] Die Ausgabe o berechnet sich mit der Aktivierungsfunktion φ als $o = \varphi(\sum_n (x_n \cdot w_n))$.[9]

Ein künstliches neuronales Netz besteht aus mehreren Neuronen in mindestens zwei Schichten. Die erste Schicht ist die Eingabeschicht. Die Neuronen der Eingabeschicht werden mit den Eingabedaten aktiviert. Oft ist es sinnvoll, vor jedes Neuron der Eingabeschicht jeweils ein weiteres Neuron mit der Ausgabe 1 zu schalten. Diese ,,nullte" Schicht, die aus den Bias besteht, ist ein technischer Trick, der verhindert, dass ein Eingabedatum von 0 das Eingabeneuron unbenutzbar macht, weil so ein Nullprodukt entsteht: $net = 0 \cdot w_n = 0$. Ein künstliches neuronales Netz würde eine Eingabe mit 0 als unwichtig werten, was nicht im Sinne des Entwicklers ist.

Es existieren verschiedene Varianten künstlicher neuronaler Netze. Die einfachste Variante ist das Feedforward-Netz. In einem Feedforward-Netzwerk sind alle Neuronen der einen Schicht vollständig und ausschließlich mit den Neuronen der nächsten Schicht verbunden.[10] Daneben existieren rekurrente künstliche neuronale Netze oder andere Netztypen, bei denen Neuronen mit sich selbst oder unter den Schichten verbunden sind, was dazu führt, dass das Netz zum Beispiel ein Kurzzeitgedächtnis bekommt. In dieser Facharbeit wird es um ein dreischichtiges Feedforward-Netz gehen, da es der simpelste Netztyp ist und für die Anwendung ausreicht. Eine Erläuterung größerer Netze würde über den Umfang dieser Facharbeit hinausgehen.

Feedforward-Netze können einschichtig oder mehrschichtig sein. Sind sie einschichtig, ist die Eingabeschicht direkt mit der Ausgabeschicht verbunden. In der Ausgabeschicht stellen die Ausgaben der Ausgabeneuronen die Ausgaben des Netzes dar. Mehrschichtige Feedforward-Netze besitzen zusätzlich noch ein oder mehrere versteckte Schichten zwischen der Eingabe- und der ausgabeschicht. In Abbildung 3 ist ein Feedforward-Netz mit drei Eingabeneuronen (rot), vier versteckten Neuronen (braun) und einem Ausgabeneuron (grün) schematisiert.

Möchte man nun die Ausgabe des Netzes bei gegebenen Eingaben berechnen, so

[8] Vgl. Belavkin, Roman V: Lecture 11: Feed-Forward Neural Networks. In: A Model of A Single Neuron, S. 3–5, http://eis.mdx.ac.uk/staffpages/rvb/teaching/BIS3226/hand11.pdf, 17. 03. 2010, Stand: 12. März 2016.

[9] Vgl Gershenson, Carlos: Artificial Neural Networks for beginners. S. 3ff. http://arxiv.org/pdf/cs/0308031v1, 20. 08. 2003, Stand: 12. März 2016.

[10] Vgl. Boersma, Paul: Feedforward neural networks 1. In: What is a feedforward neural network, http://www.fon.hum.uva.nl/praat/manual/Feedforward_neural_networks_1__What_is_a_feedforward_ne.html, 26. 04. 2004, Stand: 12. März 2016.

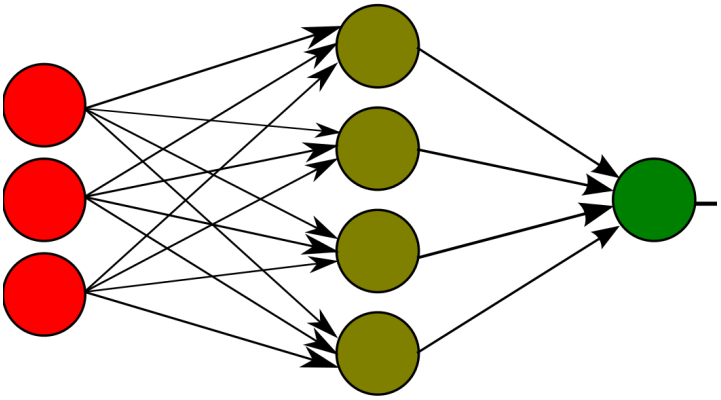

Abbildung 3: Schema eines mehrschichtigen Feedforward-Netzes[11]

könnte man die Ausgabe jedes Neurons einzeln berechnen. Um das zu vereinfachen, wird aber jede Schicht durch einen Vektor dargestellt, der so viele Dimensionen wie die Schicht Neuronen hat. Die Gewichte befinden sich in einer Matrix, die in der ersten Dimension die Anzahl an Neuronen der betroffenen Schicht hat, und als zweite Dimension die Anzahl Neuronen der nächsten Schicht. Nimmt man das Schema als Beispiel, ergibt sich für die Eingabeschicht folgender Vektor und folgende Gewichtsmatrix:

$$eingabe = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \qquad gewichte_1 = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \\ w_{4,1} & w_{4,2} & w_{4,3} \end{bmatrix}$$

Multipliziert man $eingabe \cdot gewichte_1$, erhält man die Netzeingabe für die versteckte Schicht als vierdimensionalen Vektor. Für die Ausgabe der versteckten Schicht wird für jedes Element des Vektors die Aktivierungsfunktion berechnet. Anschließend wird die Ausgabe der versteckten Schicht mit der Gewichtsmatrix der nächsten Schicht multipliziert und man erhält im Schema einen eindimensionalen Vektor. Für diesen Vektor wird wieder die Aktivierungsfunktion der Ausgabeschicht berechnet das Ergebnis wird als Ausgabe des gesamten Netzes betrachtet. Das Wissen eines künstlichen neuronalen Netzes ist auch in den Neuronenverbindungen, also in den Gewichten, gespeichert. Es gibt zwei Methoden, ein künstliches neuronales Netz zu trainieren. Die eine ist das unbeaufsichtigte Lernen und funktioniert wie beim biologischen Vorbild durch die Benutzung der Gewichte. Die andere, weit häufiger benutzte Methode ist das beaufsichtigte Lernen. Beim beaufsichtigten Lernen wird das künstliche neuronale Netz

[11] Sky99, Wikimedia Commons: Struktur eines klassischen Multi-Layer-Perzeptrons, https://commons.wikimedia.org/wiki/File:MultiLayerPerceptron.svg, 15. 04. 2012, Stand: 12. März 2016

zunächst mit Eingaben gefüttert, zu denen es bekannte Ausgaben ausgeben soll. Die Gewichte des Netzes werden so angepasst, dass die Zielausgabe möglichst gut erreicht wird. Ist eine Menge von Eingabe- und Zieldaten gegeben, macht es mehr Sinn, beaufsichtigt zu Lehren, da unbeaufsichtigtes Lernen aufwändiger ist, länger dauert und sich das Ergebnis nicht kontrollieren lässt. In der noch folgenden Anwendung ist das Beispiel so gewählt, dass es beaufsichtigt gelehrt werden kann. Um möglichst effektiv beaufsichtigt zu lehren, werden die Gewichte anfangs auf zufällige Werte gesetzt. Im anschließenden Training werden sie so modifiziert, dass der durch die Kostenfunktion, oder auch Fehlerfunktion berechnete Fehler so gering wie möglich wird. Nach dem Training folgt der Test mit Eingabedaten, zu denen die gewünschten Ausgabedaten ebenfalls bekannt sind. Der Test zeigt, ob das künstliche neuronale Netz die Muster in den Eingabedaten erkannt hat und damit in der Lage ist, bei unbekannten Daten zuverlässige Einschätzungen zu treffen.[12] Es gibt zwei Szenarien für die Ergebnisse des Tests: Das neuronale Netz kann überangepasst sein, was bedeutet, dass es zwar trainierte Daten akkurat verarbeitet, bei untrainierten Daten aber absurde Ergebnisse liefert. Das neuronale Netz kann unterangepasst sein, was dazu führt, dass es die Trainingsdaten nicht gut genug gelernt hat und weder bei den Trainings- noch bei den Testdaten gute Ausgaben liefert. Das Ziel des Lehrers ist es, das Netz durch geschickte Netzkonfiguration so zu trainieren, dass es möglichst wenig über- und unterangepasst ist.

Bei einem Feedforward-Netz bietet sich eine Lerntechnik an, die Fehlerrückführung genannt wird. In einem künstlichen neuronalen Netz ist jedes Neuron an dem Fehler, den das Netz erzeugt, beteiligt. Im Bewusstsein dieser Tatsache funktioniert die Fehlerrückführung, die den Fehler eines einzelnen Neurons mit dem Gradientverfahren berechnet und die Gewichte dieses Neurons verändert, sodass der Fehler geringer wird. Um das Gradientverfahren anwenden zu können, muss die Aktivierungsfunktion differenzierbar und kontinuierlich sein. Sei also die Aktivierungsfunktion eine Sigmoidfunktion:

$$o = \varphi(net) = \frac{1}{1 + e^{-net}}$$

Der Fehler sei berechnet als die Differenz zwischen tatsächlicher und gewünschter Gesamtausgabe: $\Delta = t - o$. Die Deltaregel der Fehlerrückführung zur Berechnung des Fehlers eines einzelnen Neurons lautet $\delta = l \cdot \Delta \cdot \varphi'(net)$. l ist das Lerntempo, eine kleine Zahl, oft 0.01. Sie spielt bei der Über- und Unteranpassung eine große Rolle. Die Ableitung der Sigmoidfunktion ist $\varphi(net)' = net(1 - net)$. Nach dem Einsetzen ergibt sich $\delta = l \cdot (t - o) \cdot (1 - net)$. Daraus sollen nun die Gewichtsveränderungen zwischen zwei Neuronen berechnet werden. Die Veränderung $\Delta w_{j,k}$ ist proportional zum Fehler

[12] Vgl. Belavkin, Roman V: Lecture 11: Feed-Forward Neural Networks. In: Training algorithms, S. 9, http://eis.mdx.ac.uk/staffpages/rvb/teaching/BIS3226/hand11.pdf, 17. 03. 2010, Stand: 12. März 2016.

des Neurons j und abhängig von der Aktivierung des Neurons k. Um das Gewicht zwischen k und j nun anzupassen, wird die Differenz berechnet: $\Delta w_{k,j} = \delta_k \cdot x_k$. In einer Eingabeschicht ist x_k eine Eingabe, in anderen Schichten die gewichtete Summe net. Die Gewichtsveränderung zwischen zwei Neuronen in einem Feedforward-Netz mit der Sigmoidfunktion als Aktivierungsfunktion in einer Ausgabeschicht berechnet sich also:

$$\Delta w_{k,j} = l \cdot (t_k - o_k) \cdot (1 - net_k) \cdot x_j$$

Es besteht nun das Problem, dass der Fehler einer versteckten Schicht oder einer Eingabeschicht vom Fehler der nächsten Schicht abhängig ist. Indem man die Gewichtsveränderung nicht mit der Ableitung der Aktivierungsfunktion, sondern mit der gewichteten Summe der Fehler der nächsten Schicht berechnet, löst man dieses Abhängigkeitsproblem:

$$\Delta w_{k,j} = l \cdot (t_k - o_k) \sum_l w_{l,k} \cdot \delta_l \cdot x_j$$

Das Netz wird so von hinten nach vorne optimiert, daher kommt der Name Fehler*rückführung*.[13]

2 Anwendung auf die Boston-Datenreihe

Mit künstlichen neuronalen Netzen können Probleme gelöst werden, zu denen keine algorithmische Lösung bekannt ist. Feedforward-Netze eignen sich, um zum Beispiel Eingabedaten zu klassifizieren, da sie in der Lage sind, Rauschen zu filtern. Bilderkennung und künstliche Intelligenz sind häufige Anwendungsgebiete, oder Voraussagen und Modellierungen.[14] Mit voraussagenden künstlichen neuronalen Netzen lassen sich zum Beispiel Entscheidungen im Aktienhandel treffen.[15] Neben der Prognose der Veränderungsrichtung von Aktienkursen, der Prognose von Wechselkursen, der kurzfristigen Prognose von Aktienkursen, werden mit künstlichen neuronalen Netzen auch Kreditwürdigkeitsanalysen getätigt.[16]
Um herauszufinden, wie ein künstliches neuronales Netz eine Voraussage trifft, wird

[13] Vgl. Papagelis, Anthony J. und Kim, Dong Soo: Multi-Layer Perceptron. In: Backpropagation, https://www.cse.unsw.edu.au/~cs9417ml/MLP2/BackPropagation.html#BackPropagation, 28. 08. 2004, Stand: 12. März 2016.

[14] Vgl. Belavkin, Roman V: Lecture 11: Feed-Forward Neural Networks. In: Applications, S. 10f, http://eis.mdx.ac.uk/staffpages/rvb/teaching/BIS3226/hand11.pdf, 17. 03. 2010, Stand: 12. März 2016.

[15] Vgl. Facklam, Yannik: Entwicklung und Analyse eines Intraday-Marktpreismodells für liquide US-Aktien mit Künstlichen Neuronalen Netzen. S. 79, http://www.iwi.uni-hannover.de/fileadmin/wirtschaftsinformatik/Abschlussarbeiten/MA_Facklam_Y._K..pdf, 27. 09. 2013, Stand: 12. März 2016.

[16] Vgl. Schneider, Bernd: Neuronale Netze für betriebliche Anwendungen. Anwendungspotentiale und existierende Systeme. In: Datenanalysen mit neuronalen Netzen, S. 14ff. https://www.wi.uni-muenster.de/sites/default/files/publications/arbeitsberichte/ab22.pdf, 11/1993, Stand: 12. März 2016.

das von mir in der Programmiersprache Python geschriebene künstliche neuronale Netz auf ein Problem der sogenannten Boston-Datenreihe trainiert und angewendet. Sie ist erhältlich über die Internetseite des CENTER FOR MACHINE LEARNING AND INTELLIGENT SYSTEMS. Die Datenreihe wurde 1978 erstellt von D. Harrison und D. L. Rubinfeld und enthält mittlere Hauspreise und Umgebungsfaktoren von Vororten Bostons. Die Daten sind vom U. S. CENSUS SERVICE erhoben worden. Die Boston-Datenreihe wird in vielen Programmiereinführungen mit dem Fokus auf maschinelles Lernen verwendet, um die Funktionsweise einer linearen Regression an einem einfachen Beispiel zu demonstrieren. In dieser Arbeit wird die Datenreihe mit Hilfe eines künstlichen neuronalen Netzes untersucht.

2.1 Aufbau und Ziel der Datenreihe

Die Datenreihe enthält 506 Elemente mit jeweils 14 Attributen.[17]

Attribut	Beschreibung	Beispielwert
`CRIM`	Die Prokopfkriminalitätsrate der Stadt	0.1415
`ZN`	Anteil an Parzellen, die für Wohngebiete markiert sind und größer als 25000 Quadratfuß	0
`INDUS`	Anteil an Grundbesitzen ohne Einzelhandelsunternehmen pro Stadt	6.91
`CHAS`	Charles-River-Variable, die 1 ist, wenn das Gebiet an einen Fluss grenzt, sonst 0	0
`NOX`	Stickoxidkonzentration in parts per 10 million	0.448
`RM`	Durchschnittliche Anzahl an Räumen pro Haus	6.169
`AGE`	Anteil an bewohnten Gebäuden, die vor 1940 gebaut wurden	6.6
`DIS`	Gewichtete Distanz zu fünf Arbeitszentren Bostons	5.7209
`RAD`	Index der Erreichbarkeit von Einfallsstraßen	3
`TAX`	Vollwert des Eigentumssteueranteils pro 10000 Dollar	233
`PTRATIO`	Verhältnis Schüleranzahl zu Lehreranzahl pro Stadt	17.9
`B`	Der Wert $1000 \cdot (B_k - 0.63)^2$, wobei B_k der Anteil an Schwarzen ist	383.37
`LSTAT`	Prozent der Unterschicht pro Stadt	5.81
`MEDV`	Mittelwert bewohnter Häuser in 1000 Dollar	25.3

Nach einem Training soll ein Algorithmus den `MEDV`-Wert einer Stadt durch die gegebenen anderen 13 Attribute möglichst genau berechnen. Mit der Boston-Datenreihe lassen sich verschiedene Lösungsmethoden für ähnliche Probleme vergleichen. Ein

[17] Lichman, M.: UCI Machine Learning Repository. In: The Boston Housing Dataset, übersetzt, https://archive.ics.uci.edu/ml/datasets/Housing, 10. 10. 1996, Stand: 12. März 2016.

Lösungsalgorithmus, der eine lineare Regression verwendet, ist in den Anlagen beigefügt, liegt aber nicht im Fokus dieser Arbeit.

2.2 Implementierung

2.2.1 Grundlage

Um ein künstliches neuronales Netz zu entwickeln, wird Tensorflow verwendet. Tensorflow ist eine von Google entwickelte und genutzte Bibliothek für die Programmiersprache Python, mit der Berechnungen schnell und effizient durchgeführt werden können. Python ist eine einfache und trotzdem schnelle Programmiersprache, die auf vielen Plattformen läuft. Sowohl Tensorflow als auch Python sind Open Source Software und damit frei nutzbar.
Verstanden, wie die Voraussage eines künstlichen neuronalen Netzes funktioniert, wird am besten anhand der Untersuchung eines funktionsfähigen Netzes. Mit Tensorflow und Python habe ich ein Feedforward-Netz erschaffen, welches in der Lage ist, das Problem der Boston-Datenreihe zu lösen. In der Implementierung findet die zuvor erklärte Mathematik Anwendung. Nach Übertragung der Formeln in algorithmische Logik werden die Ergebnisse der Umsetzung untersucht. Die Erklärung ist auf die Lösung des Problems der Boston-Datenreihe bezogen, die Umsetzungen lassen sich auf beliebige Probleme, die mit dreischichtigen Feedforward-Netzen gelöst werden können, übertragen.
Um das Training zu erleichtern, werden vor den Berechnungen alle Eingangsdaten und Zieldaten skaliert, sodass sie sich im Intervall $[0; 1]$ befinden. Die Anpassung beschleunigt den Lernvorgang. Die Eingangsdaten sind 506 Objekte mit 13 Attributen. Die erwarteten 506 Zieldaten sind die mittleren Hauspreise. Aus einem Eingabeobjekt wird ein Eingabevektor erstellt, der in der Programmierung als Liste gesehen wird. Die Listen enthalten die Attributenwerte in der oben aufgezählten Reihenfolge. Eingangsdaten und zugehörige Ausgangsdaten werden in zwei Sets aufgeteilt. Ein Set, bestehend aus zufälligen 85% der Eingangs- und zugehörigen Zieldaten, wird zum Training verwendet. Mit dem anderen Set wird das künstliche neuronale Netz bewertet. Das implementierte künstliche neuronale Netz besteht aus drei Schichten. In der ersten Schicht befinden sich die 13 Eingabeneuronen, von denen jedes ein Element der Eingabeliste als Eingabe erhält. Die zweite, versteckte Schicht, besteht aus 9 Neuronen. Die Zahl ist willkürlich gewählt und orientiert sich an der Faustformel, die Anzahl auf ungefähr zwei Drittel der Summe der Eingangs- und Ausgangsneuronen festzulegen, was in den Tests gute Ergebnisse bringt. Die Ausgabeschicht enthält ein einzelnes Neuron, dessen Ausgabe möglichst nahe an den Zielwert kommen soll.
Die Eingabeneuronen werden mit den 13 Eingangsdaten gefüttert und zu jeder Eingabe wird das Bias addiert. Diese Werte werden mit den Gewichten, die die erste und

die zweite Schicht verbinden, multipliziert. Die zweite Schicht enthält 9 Neuronen, sodass die Gewichtsmatrix die Dimensionen 13×9 hat. Die Aktivierungsfunktion wird auf jedes Element der Netzeingabe angewendet.

```
vecVersteckt = sigmoid((vecEingaben + vecBias) * matGewichteEingabe)
```

Der erhaltene Vektor hat neun Dimensionen, beziehungsweise die erhaltene Liste hat eine Länge von neun, pro verstecktem Neuron ein Element. Da das Produkt aus Vektor und Matrix ein Vektor mit den gewichteten Summen als Elementen ist, muss nicht zusätzlich eine Summe berechnet werden. Die Ausgaben der versteckten Schicht werden mit der zweiten Gewichtsmatrix multipliziert. Diese hat die Dimensionen 9×1. Wieder wird für den Ergebnisvektor der Dimension 1 die Sigmoidfunktion angewandt. Die Ausgabe dieses einen Ausgabeneurons ist die Gesamtausgabe des Netzes.

```
scalarAusgabe = alsSkalar(sigmoid(vecVersteckt * matGewichteAusgabe))
```

Die Fehlerfunktion ist die Differenz zwischen der Ausgabe und dem Ziel.

```
scalarFehler = scalarZiel - scalarAusgabe
```

Es wäre für den Computer uneffizient, nur eine Liste von Eingaben mit einem Ziel auf einmal zu berechnen. TensorFlow ist darauf ausgelegt, viele Berechnungen auf einmal durchzuführen, und braucht wesentlich weniger Zeit für eine große Berechnung als für viele kleine. Um das auszunutzen, werden bis zu 50 Eingabeobjekte und 50 Zieldaten gleichzeitig verarbeitet. So wird der Vektor der Quelldaten zu einer Matrix der Dimension 50×13 und das Zielskalar zu einem 50-dimensionalen Vektor. Für die Berechnung ergibt sich kein großer Unterschied, aber die Fehlerfunktion wird angepasst. Es wird die Summe aller 50 Differenzen genommen. Da es passieren kann, dass einzelne große Fehlerdifferenzen in der Summe 0 ergeben, wird der Betrag der Differenzen summiert. So verändert sich der Pseudocode zu:

```
matBias = dupliziere(vecBias, 50)
matVersteckt = sigmoid((matEingaben + matBias) * matGewichteEingabe)
vecAusgabe = alsVec(sigmoid(matVersteckt * matGewichteAusgabe))
```

Jede Gruppe von Eingabeobjekten und Zieldaten wird 100 mal trainiert, um dem Optimierungsalgorithmus genug Gelegenheit zum Anpassen der Netzparameter zu geben. Wird ein neuronales Netz zu wenig trainiert, werden im Test ungenaue Ergebnisse erzielt, es ist unterangepasst; wird es zu viel trainiert, ist das Netz zu sehr auf die Trainingsdaten fixiert und kann auf die Testdaten nicht eingehen, es ist überangepasst. Um das Risiko der Überanpassung zu minimieren, wird während des gesamten Trainings das Lerntempo der Fehlerrückführung verringert, sodass die Optimierung bei unbekannten Daten die Parameter langsamer verändert. Es startet bei dem Wert 0.1

und fällt in jedem weiteren Lernschritt exponentiell. Das Lerntempo eines Schrittes berechnet sich als $\text{aktuellesLerntempo} = \text{lerntempo} \cdot \text{falltempo}^{\text{schritt}/\text{fallschritt}}$. Die Variable schritt wird während des Lernens hochgezählt.[18]

2.2.2 Programmcode

Das Programm ist komplett in Eigenentwicklung entstanden. Der vollständige Quellcode des Programms befindet sich als Konsolenanwendung in den Anlagen. Umgesetzt in Python-Code besteht das Programm aus diesen wesentlichen Elementen:

```
# Vereinfachte Darstellung der Implementierung eines Feedforward-Netzes
# mit einer versteckten Schicht.
#
# Copyright (c) 2016, schneefux
#
# Parameter der Netzkonfiguration:
# Die Anzahl an Eingabeneuronen, an versteckten Neuronen
# und an Ausgabeneuronen
IN = 13
HIDDEN = 9
OUT = 1
# Die anfängliche Lerngeschwindigkeit und die Parameter
# für die Berechnung der fallenden Geschwindigkeit
LEARNINGRATE = 0.1
DECAYRATE = 0.95
DECAYSTEPS = 10

# Die Eingabeschicht
with tf.name_scope('input_layer'):
    # Die placeholder werden später von Tensorflow mit den Quelldaten
    # und den gewünschten Zieldaten gefüllt.
    matInput = tf.placeholder(tf.float32, [None, IN], name='input')
    vecTarget = tf.placeholder(tf.float32, [None], name='target')

    # Der Vektor Bias wird als Variable mit zufälligen Werten
    # initialisiert und so vermehrt, dass eine Matrix entsteht,
    # die auf die Eingaben addiert wird.
    scalarNumelements = tf.shape(vecTarget, name='batch_length')
    vecBias = tf.Variable(tf.random_uniform([IN], -1.0, 1.0), name='bias')
    vecsBias = tf.tile(vecBias, scalarNumelements)
    vecsBias = tf.reshape(vecsBias, [-1, IN])
```

[18] Vgl. Tensorflow, Google Brain Team: Open Source Library Tensorflow. In: Python API documentation, https://www.tensorflow.org/versions/master/api_docs/python/train.html#exponential_decay, 16. 02. 2016, Stand: 12. März 2016.

```
    matBiasedInput = tf.add(matInput, vecsBias, name='biased_input')

# Die versteckte Schicht
with tf.name_scope('hidden_layer'):
    # Die Gewichtsmatrix wird als Variable
    # mit zufälligen Werten initialisiert,
    matWeights = tf.Variable(tf.random_uniform([IN, HIDDEN], -1.0, 1.0),
                             name='weights')
    # mit der Eingabe+Bias multipliziert,
    matLayerin = tf.matmul(matBiasedInput, matWeights, name='layer_input')
    # und durch die Sigmoidfunktion geführt.
    matLayerout = tf.sigmoid(matLayerin, name='layer_output')

# Die Ausgabeschicht
with tf.name_scope('output_layer'):
    # Wie bei der versteckten Schicht.
    # Matrix mit zufälligen Werten initialisieren,
    matWeightsHidden = tf.Variable(
            tf.random_uniform([HIDDEN, OUT], -1.0, 1.0),
            name='hidden-weights')
    # mit der Ausgabe der letzten Schicht multiplizieren,
    matLayerinHidden = tf.matmul(matLayerout, matWeightsHidden,
                                 name='hidden_input')
    # in die Sigmoidfunktion füttern
    matLayeroutHidden = tf.sigmoid(matLayerinHidden,
                                   name='hidden_output')
    # und anschließend von einer n*1-Matrix
    # in einen n-Dimensionalen Vektor umwandeln.
    vecLayeroutHidden = tf.reshape(matLayeroutHidden, [-1])

# Die Fehlerfunktion
with tf.name_scope('cost'):
    # Der Fehler ist die Summe der Beträge aller Differenzen
    # zwischen Zielwerten und ermittelten Werten.
    vecDifference = tf.sub(vecTarget, vecLayeroutHidden, name='diff')
    scalarError = tf.reduce_sum(tf.abs(vecDifference))

# Die Trainingsalgorithmik
with tf.name_scope('train'):
    # Der Zähler 'global_step' zählt die Trainingsschritte hoch,
    # in den Diagrammen die X-Achse.
    global_step = tf.Variable(0, trainable=False)
    # Das Lerntempo wird berechnet
```

```
learning_rate = tf.train.exponential_decay(LEARNINGRATE,
                                           global_step, DECAYSTEPS,
                                           DECAYRATE,
                                           staircase=True)
# Das Gradientverfahren wird mit dem
# durch die Fehlerfunktion gefundenen
# Fehler und der eben berechneten learning rate
# angewiesen, den Fehler zu verringern.
train_step = tf.train.GradientDescentOptimizer(
    learning_rate).minimize(scalarError, global_step=global_step)
```

2.3 Evaluation verschiedener Netzkonfigurationen

Das Problem, dass das implementierte künstliche neuronale Netz löst, lässt sich auch mit einer linearen Regression lösen. Um die Genauigkeit einer Vorhersage einer linearen Regression und der künstlicher neuronaler Netze zu vergleichen, kann man den durchschnittlichen quadratischen Fehler, englisch mean squared error oder kurz MSE der Testdaten mit der Formel $\text{mse} = (\text{ziel} - \text{vorhersage})^2$ berechnen. Bei einer linearen Regression[19] erhält man einen Wert zwischen 10 und 30. Das künstliche neuronale Netz konnte in den Tests Werte vorhersagen, die genauso gut wie die einer linearen Regression waren oder noch besser.
In der Tabelle ist ein Vergleich zwischen fünf mittleren Hauspreisen, die das künstliche neuronale Netz vorhergesagt hat, und den tatsächlichen Werten. Die Differenzen sind größtenteils kleiner als 1000$.

Tatsächlicher Wert	Vorhergesagter Wert	Differenz
23.6	28.05794144	5
32.4	31.51463318	< 1
13.6	14.05183411	< 1
22.8	23.38663673	< 1
16.1	16.86310959	< 1

Nun wird analysiert, wie sich unterschiedliche Konfigurationen des neuronalen Netzes auf die Vorhersagegenauigkeit auswirken, um besser zu verstehen, wie es funktioniert. Für alle folgenden Tests werden jeweils die gleichen Eingangs- und Zieldaten verwendet, damit das Ergebnis vergleichbar ist. Die obere Tabelle ist mit Ergebnissen der Konfiguration, wie sie am Anfang des Codes der Implementierung beschrieben ist, entstanden. Das künstliche neuronale Netz wurde in 50er-Paketen 100 mal trainiert. Es ergab sich ein MSE von 11.44.
In der Abbildung 4 ist dargestellt, wie sich der Fehler der Fehlerfunktion im Verlaufe des Trainings ändert. Nach jeweils 100 Schritten gibt es einen Sprung des Fehlers,

[19] Ein Programm zur Berechnung der linearen Regression befindet sich in den Anlagen.

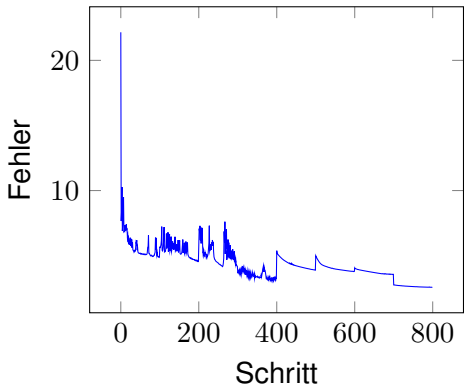

Abbildung 4: Fehler im Verlauf des Trainings

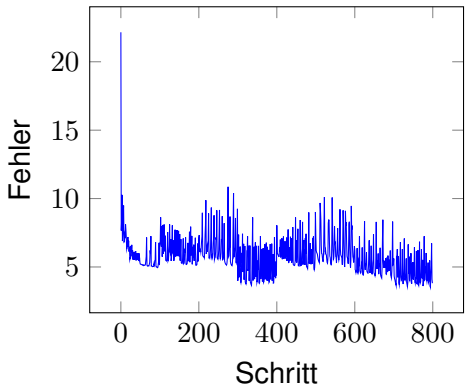

Abbildung 5: Fehler mit einem fixen Lerntempo

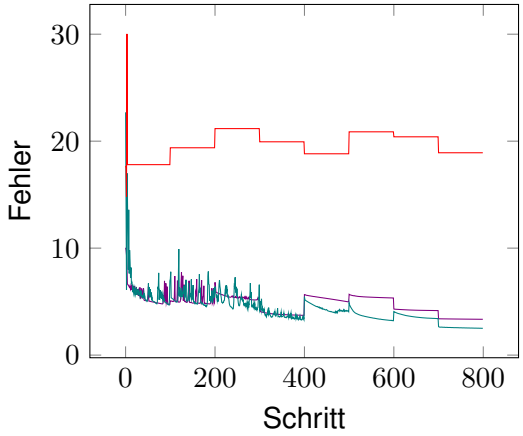

Abbildung 6: Fehler bei 1, 20, 200 versteckten Neuronen

der anschließend geglättet wird. Dieser Sprung entsteht, weil das künstliche neuronale Netz dann mit neuen, unbekannten Daten trainiert wird und sich langsam anpasst. Zu Beginn ist der Fehler am höchsten, da die zufälligen Gewichte und Bias das Netz Werte vorhersagen lassen, die weit an dem gewünschten Wert vorbei gehen. Die Optimierungsfunktion gleicht das schnell aus.

2.3.1 Auswirkung des Lerntempos und der Anzahl versteckter Neuronen

Abbildung 5 zeigt, wie sich der Fehler über die Zeit verhalten würde, wäre das Lerntempo fix auf 0.1 und würde nicht langsam absinken. Anders als in 4 sieht man keine Glättung der Fehlerkurve gegen Ende des Lernvorgangs, stattdessen wird deutlich, dass das Netz der Überanpassung unterliegt, weil es sich nicht richtig an die Trainingsdaten anpasst und der Fehler sehr stark schwankt. Die Gradientfunktion verfehlt das Fehlerminimum. Der MSE der Testdaten beträgt etwa 32 und ist damit zu hoch, als dass die Ergebnisse des Netzes nutzbar wären. Ein künstliches neuronales Netz lernt also besser, je entspannter es sich nach einem längeren Training gegenüber neuen Daten verhält, und trifft dann bessere Voraussagen.
Die Anzahl versteckter Neuronen hat großen Einfluss auf die Genauigkeit der Vorher-

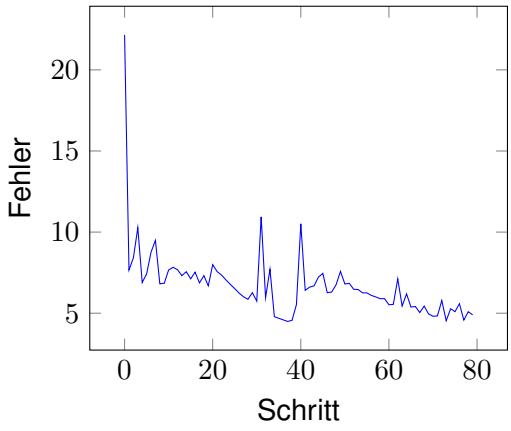

Abbildung 7: Fehler bei 10 Trainingsschritten

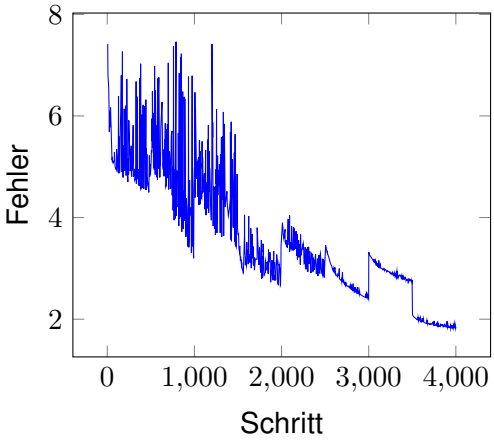

Abbildung 8: Fehler bei 500 Trainingsschritten

sagen. Wie man in Abbildung 6 sieht, ist die Optimierungsfunktion mit 200 versteckten Neuronen überfordert, kann sich aber mit 20 versteckten Neuronen besser auf die Boston-Daten anpassen, als mit nur einem. 200 versteckte Neuronen sind zu viele, als dass das Netz noch lernen könnte, aber mit einem versteckten Neuron ist es noch nicht ausgereizt. Hat man nur ein verstecktes Neuron, ist der Unterschied zu keiner versteckten Schicht nur sehr gering. Künstliche neuronale Netze treffen bessere Voraussagen, wenn die Eingangsdaten durch eine versteckte Schicht gehen, die nicht zu komplex ist.

2.3.2 Auswirkung der Anzahl Trainingsschritte

Die Folgen von Unteranpassung sieht man, wenn man die Anzahl an Trainingsschritten zu stark verringert. Die Optimierungsfunktion bekommt zu wenige Gelegenheiten, das künstliche neuronale Netz an die Trainingsdaten anzupassen. In Abbildung 7 ist es zu wenig auf die Trainingsdaten angepasst, was man daran sehen kann, dass der Fehler gegen Ende des Trainings immer noch über 5 liegt. 500 Trainingsschritte, gezeigt in Abbildung 8, würden zu einer Überanpassung führen, würde das Lerntempo nicht schnell genug sinken. Zu Anfang des Trainings mit 500 Schritten sieht man, wie der Fehler noch stark springt, aber dann gegen Ende gleichmäßig sinkt, was zu Ergebnissen führt, die mit der Anfangskonfiguration vergleichbar sind.
Ein künstliches neuronales Netz trifft bessere Voraussagen, wenn es lange genug trainiert wurde.

2.3.3 Anpassung des Lerntempos und der Anzahl versteckter Neuronen

Nun wird das sinkende Lerntempo untersucht, da es offensichtlich eine große Rolle zur Verhinderung von Überanpassung spielt und die Qualität der Voraussagen damit beeinflusst. In den vorherigen Tests wurden, wenn nicht anders angegeben, die Trai-

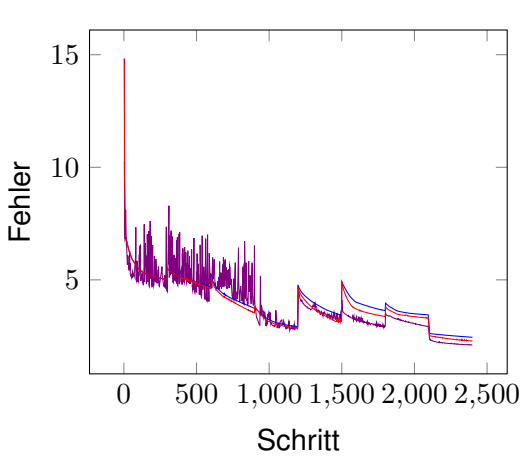

Abbildung 9: Ergebnisse der Optimierungen für 300 Trainingsschritte

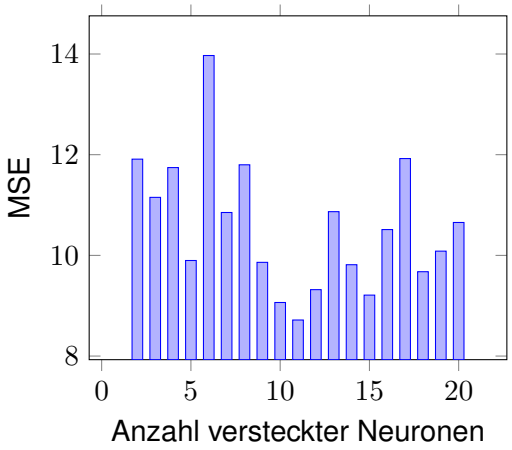

Abbildung 10: Mean squared error bei verschiedenen Anzahlen versteckter Neuronen

ningsdaten 100 mal trainiert. Das Lerntempo ist berechnet worden als $0.1 \cdot 0.95^{\frac{s}{10}}$ für jeden Schritt s. Wie in dem letzten Test sichtbar war, ist es besser, etwas länger zu trainieren, als zu wenig. Also wird die Anzahl an Trainingsschritten des Testnetzes auf 300 erhöht. Damit das Lerntempo nicht zu schnell sinkt, wird der Fallschritt verdreifacht. Der Fehlergraph ist in Abbildung 9 violett, im Test ergibt sich ein MSE von 8.61. Der Fehler springt am Anfang sehr stark und ändert sich gegen Ende kaum noch. Verringert man das Anfangslerntempo auf 0.01 und erhöht man das Falltempo auf 0.9 und den Fallschritt auf 100, ergibt sich der wesentlich ruhigere blaue Graph und ein MSE von 10.39. Bei einem Falltempo von 0.96 und einem Fallschritt von 50 ist das Netz am Ende noch etwas anpassungsfreudiger – dargestellt im roten Graphen – und erzielt einen MSE von 9.86. Auch wenn der MSE damit etwas höher als bei der Konfiguration zum violetten Graphen ist, wird sie bevorzugt, da sie in den letzten beiden Blöcken sich noch stärker anpasst, und durch diese Anpassungsfähigkeit besser bei anderen Zusammenstellungen der Trainingsdaten abschneiden würde.

Die optimale Anzahl an versteckten Neuronen für diese Netzkonfiguration soll jetzt systematisch gefunden werden. In Abbildung 10 sind die MSE für 2 bis 20 versteckte Neuronen in der versteckten Schicht abgebildet. Der MSE eines einzelnen Neurons, 28.52, ist weit von den anderen Werten entfernt. 11 versteckte Neuronen bringen das beste Ergebnis mit einem MSE von 8.72. Diese Netzkonfiguration bringt ein um 4 Einheiten besseres Ergebnis als das einer linearen Regression mit 13.73 Einheiten.

3 Fazit

Wie trifft ein künstliches neuronales Netz eine Voraussage?

Ein künstliches neuronales Netz gewichtet Eingabedaten. Die Genauigkeit der Gewichtung hängt von der Qualität der Konfiguration und des Trainings ab. Die optima-

le Konfiguration lässt sich nicht berechnen, sondern muss ausprobiert werden. Das ,,Computergehirn" erkennt die für den Anwender verborgenen Zusammenhänge mit bestimmten Konfigurationen besser, als mit anderen. Es erklärt dem Anwender jedoch nicht den Lösungsweg. Ein künstliches neuronales Netz zeigt dem Anwender statistische Zusammenhänge, die dieser selbst interpretieren muss.

Das erklärte und implementierte künstliche neuronale Netz ist klein und einfach. Mit größeren künstlichen neuronalen Netzen, die auch unterschiedliche Netztypen kombinieren, ist es möglich, dass Computer komplizierte Probleme besser lösen, als ein Mensch es kann. Anders als bei einem Algorithmus besitzt ein künstliches neuronales Netz ein gewisses Maß an Intelligenz. Die Reaktion auf Eingabedaten kann zwar nachvollzogen, aber nicht unbedingt erklärt werden.

Es existieren verschiedene Projekte und Spielereien mit künstlichen neuronalen Netzen, die das für Computer Unmögliche möglich zu machen scheinen. Mit neuraltalk2[20] werden Bilder beschrieben, mit char-rnn[21] imitiert ein rekurrentes künstliches neuronales Netz Shakespeare. neural-storyteller[22] erzählt Geschichten angeregt durch Fotos, mit sogenannten deep convolutional neural networks überträgt ein Forscherteam den Zeichenstil berühmter Künstler auf beliebige Bilder[23].

Künstliche neuronale Netze spielen Computerspiele perfekt, wie zum Beispiel MarI/O das Spiel Super Mario[24]. Ein Google-Team hat mit der künstlichen Intelligenz AlphaGo die weltbesten Go-Spieler schlagen können[25]. Zuvor glaubte man, ein Computer wäre dem Menschen in diesem Brettspiel noch lange unterlegen, da, anders als beim Schach, die 10^{761} Zugmöglichkeiten die Kapazität jedes Rechenzentrums übersteigen und daher nicht alle vorausberechnet werden können. AlphaGo ist in der Lage, den Zug vorherzusagen, der am wahrscheinlichsten zum Sieg führt.

Das Forschungsgebiet künstlicher neuronaler Netze ist relevanter denn je, es ist zu erwarten, dass in nicht allzu langer Zeit künstliche neuronale Netze auch auf mobilen Geräten Einzug halten werden und vielleicht eines Tages das Gehirn des Menschen erweitern können.

---

[20] Karpathy, Andrej und Johnson, Justin: neuraltalk2, https://github.com/karpathy/neuraltalk2, 15. 11. 2015, Stand: 12. März 2016.

[21] Karpathy, Andrej: Andrej Karpathy blog. In: The Unreasonable Effectiveness of Recurrent Neural Networks, https://karpathy.github.io/2015/05/21/rnn-effectiveness/, 21. 03. 2015, Stand: 12. März 2016.

[22] samim: Generating Stories about Images. Recurrent neural network for generating stories about images, https://medium.com/@samim/generating-stories-about-images-d163ba41e4ed#.618pkay3f, 05. 11. 2015, Stand: 12. März 2016.

[23] Zsolnai-Fehér, Károly: Two Minute Papers. Deep Neural Network Learns Van Gogh's Art, https://www.youtube.com/watch?v=-R9bJGNHltQ, 29. 08. 2015, Stand: 12. März 2016.

[24] SethBling: MarI/O. Machine Learning for Video Games, https://www.youtube.com/watch?v=qv6UVOQ0F44, 13. 06. 2015, Stand: 12. März 2016.

[25] Silver, David und Hassabis, Demis: Google Research Blog. In: AlphaGo. Mastering the ancient game of Go with Machine Learning, http://googleresearch.blogspot.de/2016/01/alphago-mastering-ancient-game-of-go.html, 27. 01. 2016, Stand: 12. März 2016.

4 Literaturverzeichnis

- Antwerpes, Frank: DocCheck Flexikon. Das Medizinlexikon zum Medmachen. In: Rezeptor, http://flexikon.doccheck.com/de/Rezeptor, 19. 01. 2015, Stand: 12. März 2016.
- Belavkin, Roman V: Lecture 11: Feed-Forward Neural Networks. In: A Model of A Single Neuron, S. 3–5, http://eis.mdx.ac.uk/staffpages/rvb/teaching/BIS3226/hand11.pdf, 17. 03. 2010, Stand: 12. März 2016.
- Belavkin, Roman V: Lecture 11: Feed-Forward Neural Networks. In: Training algorithms, S. 9, http://eis.mdx.ac.uk/staffpages/rvb/teaching/BIS3226/hand11.pdf, 17. 03. 2010, Stand: 12. März 2016.
- Belavkin, Roman V: Lecture 11: Feed-Forward Neural Networks. In: Applications, S. 10f, http://eis.mdx.ac.uk/staffpages/rvb/teaching/BIS3226/hand11.pdf, 17. 03. 2010, Stand: 12. März 2016.
- Boersma, Paul: Feedforward neural networks 1. In: What is a feedforward neural network, http://www.fon.hum.uva.nl/praat/manual/Feedforward_neural_networks_1__What_is_a_feedforward_ne.html, 26. 04. 2004, Stand: 12. März 2016.
- Chrislb, Wikimedia Commons: Schema eines künstlichen Neurons, https://commons.wikimedia.org/wiki/File:ArtificialNeuronModel_deutsch.png, 14. 07. 2005, Stand: 12. März 2016.
- Facklam, Yannik: Entwicklung und Analyse eines Intraday-Marktpreismodells für liquide US-Aktien mit Künstlichen Neuronalen Netzen. S. 79, http://www.iwi.uni-hannover.de/fileadmin/wirtschaftsinformatik/Abschlussarbeiten/MA_Facklam_Y._K..pdf, 27. 09. 2013, Stand: 12. März 2016.
- Gershenson, Carlos: Artificial Neural Networks for beginners. S. 3ff. http://arxiv.org/pdf/cs/0308031v1, 20. 08. 2003, Stand: 12. März 2016.
- Karpathy, Andrej: Andrej Karpathy blog. In: The Unreasonable Effectiveness of Recurrent Neural Networks, https://karpathy.github.io/2015/05/21/rnn-effectiveness/, 21. 03. 2015, Stand: 12. März 2016.
- Karpathy, Andrej und Johnson, Justin: neuraltalk2, https://github.com/karpathy/neuraltalk2, 15. 11. 2015, Stand: 12. März 2016.
- Könneker, Carsten und Reichert, Uwe: Lexikon der Biologie. In: Effektorzellen, http://www.spektrum.de/lexikon/biologie/effektorzellen/20145, 1999, Stand: 12. März 2016.
- Kriesel, David: Ein kleiner Überblick über Neuronale Netze. In: Zur Geschichte Neuronaler Netze, S. 10ff. http://www.dkriesel.com/_media/science/neuronalenetze-de-zeta2-1col-dkrieselcom.pdf, 2007, Stand: 12. März 2016.

– Lichman, M.: UCI Machine Learning Repository. In: The Boston Housing Dataset, übersetzt, https://archive.ics.uci.edu/ml/datasets/Housing, 10. 10. 1996, Stand: 12. März 2016.
– Lossau, Norbert: Die Welt. In: Forscher wollen die Highways im Gehirn nachbauen, http://www.welt.de/wissenschaft/article124674193/Forscher-wollen-die-Highways-im-Gehirn-nachbauen.html, 09. 02. 2014, Stand: 13. März 2016.
– Papagelis, Anthony J. und Kim, Dong Soo: Multi-Layer Perceptron. In: Backpropagation, https://www.cse.unsw.edu.au/~cs9417ml/MLP2/BackPropagation.html#BackPropagation, 28. 08. 2004, Stand: 12. März 2016.
– samim: Generating Stories about Images. Recurrent neural network for generating stories about images, https://medium.com/@samim/generating-stories-about-images-d163ba41e4ed#.618pkay3f, 05. 11. 2015, Stand: 12. März 2016.
– Schneider, Bernd: Neuronale Netze für betriebliche Anwendungen. Anwendungspotentiale und existierende Systeme. In: Datenanalysen mit neuronalen Netzen, S. 14ff. https://www.wi.uni-muenster.de/sites/default/files/publications/arbeitsberichte/ab22.pdf, 11/1993, Stand: 12. März 2016.
– SEER, US National Cancer Institute, Dhp1080 und Unbekannt, Wikimedia Commons: Neuron, https://commons.wikimedia.org/wiki/File:Neuron_(deutsch)-1.svg, 19. 12. 2006, Stand: 12. März 2016.
– SethBling: MarI/O. Machine Learning for Video Games, https://www.youtube.com/watch?v=qv6UVOQ0F44, 13. 06. 2015, Stand: 12. März 2016.
– Silver, David und Hassabis, Demis: Google Research Blog. In: AlphaGo. Mastering the ancient game of Go with Machine Learning, http://googleresearch.blogspot.de/2016/01/alphago-mastering-ancient-game-of-go.html, 27. 01. 2016, Stand: 12. März 2016.
– Sky99, Wikimedia Commons: Struktur eines klassischen Multi-Layer-Perzeptrons, https://commons.wikimedia.org/wiki/File:MultiLayerPerceptron.svg, 15. 04. 2012, Stand: 12. März 2016.
– Tensorflow, Google Brain Team: Open Source Library Tensorflow. In: Python API documentation, https://www.tensorflow.org/versions/master/api_docs/python/train.html#exponential_decay, 16. 02. 2016, Stand: 12. März 2016.
– Weber, Michael: Webmic's Page. In: Reflexe, http://www.webmic.de/reflexe.htm, 2000, Stand: 12. März 2016.
– Zsolnai-Fehér, Károly: Two Minute Papers. Deep Neural Network Learns Van Gogh's Art, https://www.youtube.com/watch?v=-R9bJGNHltQ, 29. 08. 2015, Stand: 12. März 2016.

5 Anlagen

Vollständige Implementierung des künstlichen neuronalen Netzes

```
#!/usr/bin/python3
#
# Copyright (c) 2016, schneefux
#
from sklearn import datasets
from sklearn import preprocessing
from sklearn.cross_validation import train_test_split
import tensorflow as tf
import numpy.random
import numpy as np

# Siehe Kommentar in dem Programm der linearen Regression
SEED = 42
numpy.random.seed(SEED)
tf.set_random_seed(SEED)

# Datenreihe laden
dataset = datasets.load_boston()
data, target = dataset.data, dataset.target

# Datenwerte für ein effektiveres Lernen zwischen 0 und 1 skalieren
data_scaler = preprocessing.MinMaxScaler()
data = data_scaler.fit_transform(data)
target_scaler = preprocessing.MinMaxScaler()
target = target_scaler.fit_transform(target)

# Datenwerte in Trainings- und Testreihe aufteilen
x_train, x_test, y_train, y_test = train_test_split(
    data, target, train_size=0.85
)

# Die Netzkonfiguration
BATCH = 50
STEPS = 300
HIDDEN = 11
IN = 13
OUT = 1
LEARNINGRATE = 0.01
DECAYRATE = 0.96
DECAYSTEPS = 50
```

```
# Der Lernverlauf wird in einem individuellem Dateinamen gespeichert,
# der je nach Konfiguration variiert.
RUNNAME = 'B'+str(BATCH)+'S'+str(STEPS)+'H'+str(HIDDEN)\
        + 'LR'+str(LEARNINGRATE) + 'DR'+str(DECAYRATE)\
        + 'DS'+str(DECAYSTEPS)+'SD'+str(SEED)

sess = tf.InteractiveSession()
# Ausführliche Kommentare und ein vereinfachter Ablauf
# finden sich in Abschnitt 2.2.2 der Facharbeit.

with tf.name_scope('input_layer'):
    matInput = tf.placeholder(tf.float32, [None, IN], name='input')
    vecTarget = tf.placeholder(tf.float32, [None], name='target')
    scalarNumelements = tf.shape(vecTarget, name='batch_length')
    vecBias = tf.Variable(tf.random_uniform([IN], -1.0, 1.0), name='bias')
    summaryBias = tf.histogram_summary("bias", vecBias)
    vecsBias = tf.tile(vecBias, scalarNumelements)
    vecsBias = tf.reshape(vecsBias, [-1, IN])
    matBiasedInput = tf.add(matInput, vecsBias, name='biased_input')

with tf.name_scope('hidden_layer'):
    matWeights = tf.Variable(tf.random_uniform([IN, HIDDEN], -1.0, 1.0),
                             name='weights')
    summaryWeights = tf.histogram_summary('weights', matWeights)
    matLayerin = tf.matmul(matBiasedInput, matWeights, name='layer_input')
    matLayerout = tf.sigmoid(matLayerin, name='layer_output')

with tf.name_scope('output_layer'):
    vecTarget = vecTarget
    matWeightsHidden = tf.Variable(
            tf.random_uniform([HIDDEN, OUT], -1.0, 1.0),
            name='hidden-weights')
    summaryWeightsHidden = tf.histogram_summary('hiddenweights',
                                                matWeightsHidden)
    matLayerinHidden = tf.matmul(matLayerout, matWeightsHidden,
                                 name='hidden_input')
    matLayeroutHidden = tf.sigmoid(matLayerinHidden, name='hidden_output')
    # nur vec, wenn OUT=1
    vecLayeroutHidden = tf.reshape(matLayeroutHidden, [-1])

with tf.name_scope('cost'):
    vecDifference = tf.sub(vecTarget, vecLayeroutHidden, name='diff')
```

```
    scalarError = tf.reduce_sum(tf.abs(vecDifference), name='error')
    summaryError = tf.scalar_summary('error', scalarError)

with tf.name_scope('train'):
    global_step = tf.Variable(0, trainable=False)
    learning_rate = tf.train.exponential_decay(LEARNINGRATE,
                                               global_step, DECAYSTEPS,
                                               DECAYRATE, staircase=True)
    train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(
            scalarError, global_step=global_step)

summaries = tf.merge_all_summaries()
writer = tf.train.SummaryWriter("/tmp/boston_logs/%s" % RUNNAME,
                                sess.graph_def)
# Alle Tensorflow-Variablen müssen bis hier spezifiziert sein!
sess.run(tf.initialize_all_variables())

# Training
for count in range(0, int(len(x_train) / BATCH)):
    print("batch " + str(count))
    for i in range(0, STEPS):
        result = sess.run([summaries, train_step],
                          feed_dict={
                              matInput: x_train[count*BATCH:(count+1)*BATCH],
                              vecTarget: y_train[count*BATCH:(count+1)*BATCH]
                          })
        writer.add_summary(result[0], count * STEPS + i)

print("Training beendet")

print("Testergebnisse - Differenz zu echten Daten:")
result = sess.run([vecLayeroutHidden],
                  feed_dict={
                      matInput: x_test,
                      vecTarget: y_test
                  })
print("================================")
print("erwartete Daten:")
print("--------------------------------")
# Ausgabedaten sind skaliert, zurückskalieren
print(target_scaler.inverse_transform(y_test))
print("--------------------------------")
print("vorhergesagte Daten:")
```

```
print(target_scaler.inverse_transform(result[0]))
print("---------------------------------")
print("")
print("Differenz:")
print("---------------------------------")
# MSE aus der Differenz unskalierter Daten berechnen,
# nicht MSE aus Differenz skalierter Daten unskalieren:
# !!! (a - b) * f + o != (a * f + o) - (b * f + o)
# <=> af - bf + o     != af - bf
diff = target_scaler.inverse_transform(result[0]) -
           target_scaler.inverse_transform(y_test)
print(diff)
print("---------------------------------")
print("")
print("MSE")
# Entsprechend (af - bf + o)^2 != (af - bf)^2...
print(np.mean(diff ** 2))
```

Vollständige Implementierung einer linearen Regression

```
#!/usr/bin/python3
#
# Copyright (c) 2016, schneefux
#
from sklearn.datasets import load_boston
from sklearn.linear_model import LinearRegression
from sklearn.cross_validation import train_test_split
import numpy as np

# Ein gleicher Wert SEED bei der linearen Regression
# und bei der Implementierung des kNN bewirkt,
# dass Trainings- und Testreihe gleich sind
# und damit das Ergebnis vergleichbar.
SEED = 42
np.random.seed(SEED)

# Datenreihe laden und in Trainings- und Testreihe aufteilen
data = load_boston()
X_train, X_test, y_train, y_test = train_test_split(
    data.data, data.target, train_size=0.85)

# Die lineare Regression auf die Trainingsdaten anwenden
model = LinearRegression()
model.fit(X_train, y_train)
```

```
# Die Testdaten vorhersagen
prediction = model.predict(X_test)
# Das Quadrat aller Differenzen berechnen
mse = np.mean([
    (y_test[n] - prediction[n]) ** 2
    for n in range(len(prediction))
])

print(mse)
```

6 Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Stadt, 24. Juli 2016

SCHNEEFUX