# Modellierung, Implementierung und Strukturvergleich eigener neuronaler Netze zur Handschrifterkennung und Vergleich mit moderner Bibliothek

Facharbeit
in der Fachrichtung Informatik
des Albert-Schweitzer-Gymnasiums Kaiserslautern
am Gymnasium am Rittersberg

vorgelegt von
Boris Giba
(Betreuer: Patrick Haag)

Kaiserslautern 2019

# Abstrakt

In dieser Arbeit wird das Konzept eines künstlichen neuronalen Netzwerks erklärt. Es wird ein Feedforward-Netzwerk in Python selbstständig implementiert und mithilfe des Backpropagation-Algorithmus auf 60000 Datensätzen in Form von Abbildungen handschriftlich gezeichneter Ziffern trainiert. Es werden mehrere Netze mit verschiedenen Strukturen erstellt und auf jeweils einem Sechstel der Daten trainiert. Hierbei wird jede Netzstruktur mit je drei verschiedenen Lernraten separat trainiert. Mit der Bibliothek TensorFlow werden ebenfalls Netze mit denselben Strukturen erstellt und auch auf jenen Daten trainiert.

Es wird herausgefunden, dass größere Netze mit geringeren Lernraten eine höhere Genauigkeit erzielen als mit größeren Lernraten. Ebenfalls wird festgestellt, dass es für jedes Problem eine optimale Netzstruktur geben muss, und dass ein komplexeres Netz nicht zwingend mit einer höheren Genauigkeit verbunden sein muss. Die eigene Implementierung unterscheidet sich von der modernen Bibliothek hauptsächlich im Kriterium Geschwindigkeit (Zeitkosten).

# Inhaltsverzeichnis

# Abbildungsverzeichnis



# Tabellenverzeichnis

# Abkürzungsverzeichnis

knN: künstliches neuronales Netz(werk)

TF: TensorFlow

f.ü.: frei übersetzt

.csv: Dateiformat; wörtlich: durch Kommata getrennte Werte (eng.: *comma-separated values*)

.pkl: Dateiformat, welches von der Python-Bibliothek pickle genutzt wird

d.h.: das heißt

# 1 Motivation und Schwerpunkt

## 1.1 Motivation

Heutzutage hört man in den Medien immer mehr über „Machine Learning“. Vom Sprachassistenten bis hin zu ersten Anfängen des autonomen Fahrens findet sich dieses Themengebiet in vielen Bereichen der modernen Technologie wieder.
Angesichts der heutzutage unanfechtbaren Relevanz künstlicher Intelligenz sowie aus persönlichem Interesse soll diese Arbeit einen ersten Einblick in den weit verbreiteten Anwendungsbereich der künstlichen neuronalen Netze bieten.

## 1.2 Schwerpunkt und Ergebnis

Der Schwerpunkt dieser Arbeit liegt hierbei im Praxisabschnitt. Hier soll ein eigenes Netz modelliert, sowie implementiert werden, welches am Beispiel von Bildern, in Form von handschriftlichen Ziffern, zur Klassifikation jener trainiert und evaluiert werden soll. Ebenfalls sollen theoretische Grundlagen und Anwendungszwecke erläutert, sowie verschiedene Netzstrukturen analysiert werden. Zum Schluss wird die eigene Implementierung mit Netzen verglichen, welche mithilfe der Bibliothek TensorFlow generiert und trainiert wurden. Es gelingt, ein eigenes knN zu generieren. Nachdem das Netz mit 60,000 Bildern trainiert wurde, erreicht es eine Klassifikationsgenauigkeit von ca. 95,16 % bei dem Netz unbekannten Datensätzen.

# 2 Theoretischer Teil

## 2.1 Was ist ein neuronales Netz?

Ein [neuronales Netzwerk] ist ein […] [System], welches die Fähigkeit hat, seine strukturellen Bestandteile, bekannt als Neuronen, zu organisieren, um eine bestimmte [Aufgabe] zu erfüllen (wie beispielsweise Mustererkennung) […].[1] Heutzutage finden sie beispielsweise in der Klassifizierung von Zellen

[1] f.ü. aus: Haykin, Simon (2009): *Neural Networks and Learning Machines.* Hamilton: Pearson (S.31)

für die Krebsdiagnose oder bei der Fehleridentifizierung von Telefonschaltsystemen Anwendung.[2]

### 2.1.1 Neuronen

Man findet beispielsweise im menschlichen Gehirn Nervenzellen (oder auch Neuronen) wieder.[3] Eine Abbildung eines solchen Neurons lässt sich rechts wiederfinden. An den Dendriten werden Impulse von außerhalb aufgenommen, welche dann am Axon weitergeleitet werden können. Wird ein Impuls über das Axon übertragen, so spricht man hier auch vom Aktionspotential.[4] Die Stelle, an der eine Nervenzelle in Kontakt mit einer anderen Nervenzelle steht, kann auch als Synapse bezeichnet werden.

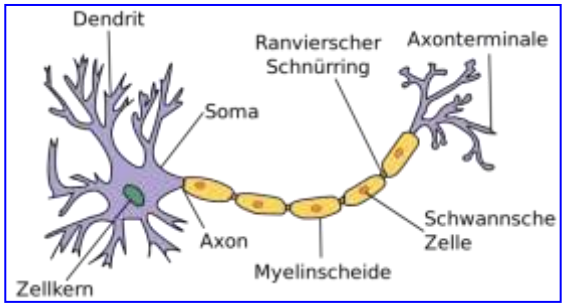



*Abbildung 1: biologisches Neuron (https://commons.wikimedia.org/wiki/File:Neuron_Hand-tuned.svg ; 10.02.2019)*

## 2.2 Künstliche neuronale Netze

Die Gestaltung eines [künstlichen] neuronalen Netzes ist motiviert durch die Analogie zum Gehirn, welches ein lebender Beweis dafür ist, dass fehlertolerante, parallelisierte Berechnung nicht nur physisch möglich, sondern auch schnell und machtvoll ist.[5] Ein knN erhält hierbei einen Datensatz bestehend aus Gleitkommazahlen als Eingabe und gibt eine einzeilige Matrix (Vektor) gleichen Zahlentyps zurück. Ein knN ist in verschiedene Schichten, welche aus Neuronen bestehen, gegliedert.

---

[2] Patterson, Dan (1997): *Künstliche neuronale Netze: das Lehrbuch*. München: Prentice Hall. (S.234ff.)

[3] siehe *https://developingchild.harvard.edu/science/key-concepts/brain-architecture/*

[4] vgl. Kramer, Oliver (2009): *Computational Intelligence*. Dortmund: Springer (S.120)

[5] f.ü. aus: Haykin, Simon (2009): *Neural Networks and Learning Machines*. Hamilton: Pearson (S.34)

### 2.2.1 Künstliche Neuronen

Künstliche Neuronen sollen ähnlich funktionieren wie ihre biologischen Vorbilder. Die Aufnahme von Impulsen/Signalen wird durch Verbindungen mithilfe von Synapsen realisiert. Eine Synapse verbindet hierbei genau zwei Neuronen miteinander. Jedes Neuron erhält also über Synapsen Werte von anderen Neuronen übergeben. Diese Werte können im reellen, binären oder bipolaren Bereich liegen. Die Synapsen besitzen hierbei ein Gewicht, welches die übertragenen Werte erniedrigt oder erhöht. Die Gewichte können im selben Definitionsbereich liegen, in der Regel werden aber reelle Zahlen aus dem Intervall [-1;1] verwendet.[6] Um zu verhindern, dass Neuronen mit dem Wert 0 passiv werden, erhält jedes Neuron ein sogenanntes Bias, welches zum gewichteten Wert addiert wird.[7] Auch hierfür wird meist eine reelle Zahl aus dem Intervall [-1;1] gewählt. Der Wert, den ein Neuron nach Verarbeitung der Inputs besitzt, wird Aktivierung genannt. Jene wird berechnet, indem der gewichtete Wert durch eine Aktivierungsfunktion Φ verarbeitet wird.[8]

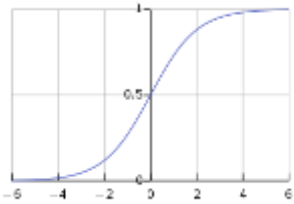



*Abbildung 2: Sigmoidfunktion (https://commons.wikimedia.org/wiki/File:Logistic-curve.svg ; 10.02.2019)*

Eine graphische Darstellung eines solchen künstlichen Neurons zur Verdeutlichung findet sich im Anhang [Seite 17] wieder. Für diese Arbeit wird ausschließlich die rechts aufgeführte Sigmoidfunktion σ mit

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

als Aktivierungsfunktion genutzt. Sie ordnet jedem reellen Eingabewert einen reellen Ausgabewert aus dem Intervall (0;1) zu und wurde in frühen Tagen künstlicher neuronaler Netze standardmäßig als Aktivierungsfunktion genutzt, ist heute allerdings veraltet. Um die Funktionsweise eines knN zu verstehen eignet sie sich jedoch nach wie vor und wird aus diesem Grund in dieser Arbeit verwendet.

---

[6] vgl. Patterson, Dan (1997): *Künstliche neuronale Netze: das Lehrbuch*. München: Prentice Hall. (S.14)

[7] vgl. Daniel Shiffman (2012): *The Nature Of Code (10.3)*

[8] vgl. Daniel Shiffman (2012): *The Nature Of Code (10.2)*

### 2.2.2 Neuronenschichten

Ein neuronales Netzwerk besteht aus Schichten von Neuronen (eng.: „*neuron layer*“). Hier werden nur Netze betrachtet, in welchen alle Neuronen einer Schicht mit allen Neuronen der benachbarten Schichten verbunden sind (eng.: „*fully connected*“)

Die erste Schicht eines knN wird auch Eingabe- oder Inputschicht (eng.: „*input layer*“) genannt. Alle Schichten eines knN zwischen der Inputschicht und der Outputschicht werden verborgene Schichten (eng.: „*hidden layer*“) genannt. Die letzte Schicht eines knN wird auch Ausgabe- oder Outputschicht (eng.: „*output layer*“) genannt.

### 2.2.3 Feedforward

Um einen Datensatz mit dem knN zu verarbeiten soll hier der Feedforward-Algorithmus genutzt werden. Zu Beginn werden die Aktivierungen der Inputschicht auf die einzelnen Werte des Datensatzes gesetzt. Daraufhin werden die Aktivierungen solange Schicht für Schicht über die Synapsen weitergeleitet, bis die letzte Schicht erreicht ist. Die Aktivierungen der Outputschicht bilden nun die Voraussage des Netzes. Bei einem Klassifizierungsproblem (beispielsweise die hier betrachtete Handschrifterkennung) hat idealerweise genau ein Neuron dieser Schicht die Aktivierung 1, alle restlichen Neuronen hingegen den Wert 0. In der Praxis kann dieser Idealfall allerdings nur sehr selten erreicht werden. Die Voraussage ist in beiden Fällen der Index des Neurons in der Ausgabeschicht, welches die höchste Aktivierung besitzt. Bei anderen Problemtypen (beispielsweise Regression) sieht die erwünschte Ausgabe meist grundsätzlich anders aus (als bei einem Klassifizierungsproblem).

### 2.2.4 Backpropagation

Die Synapsengewichte und Bias der Neuronen werden zu Beginn zufällig festgelegt. Um das Netz zu optimieren sollen die Synapsengewichte (nicht jedoch die Bias) mit dem Backpropagation-Algorithmus angepasst werden.

Mit jenem Algorithmus soll also eine Fehlerberechnung und –korrektur vorgenommen werden. Hier soll jede Ausgabe des Feedforward-Algorithmus

mit einer gewünschten Idealausgabe verglichen und Fehlerwerte für die einzelnen Neuronen der Outputschicht berechnet werden. Es findet also ein überwachtes Training (eng.: *supervised learning*) statt, da die Ausgabe des Netzes mit einer vorher bereits bekannten, idealen Ausgabe verglichen wird. Die fehlerhaften Outputwerte werden gemeinsam mit den Idealwerten dem Backpropagation-Algorithmus übergeben. Dieser passt die jeweiligen Synapsen zwischen der Outputschicht und der letzten verborgenen Schicht an, um das gewünschte Verhalten zu approximieren. Danach sollen die Fehler der vorherigen Schicht berechnet werden und die Korrektur beginnt erneut. Es wird also iterativ von der letzen bis hin zur ersten Schicht vorgegangen. Die Fehler der verborgenen Schichten werden basierend auf den Fehlern der vorausgehenden Schicht berechnet.[9] Das Ziel ist, die entstehenden Fehlerwerte zu minimieren. Das Verfahren zur Bestimmung der jeweiligen Änderungen der Synapsengewichte heißt Gradientverfahren. Es wird in dieser Arbeit genutzt, jedoch nicht im Detail betrachtet. Um mit jenem Verfahren herauszufinden, wie die jeweiligen Synapsengewichte eines Netzes angepasst werden müssen, wird wie folgt vorgegangen.

Die Änderungen aller Synapsengewichte $\Delta w$ zwischen zwei Schichten $I$ und $J$ sollen in Anbetracht der Relevanz jedes Gewichts der betrachteten Synapsen berechnet werden. Diese sollen auch abhängig von den Aktivierungen $a$ der beiden Schichten und den Fehlern $e$ der betrachteten (/vorderen) Schicht geschehen. Es werden also die Aktivierungen der vorherigen Schicht benötigt, um den bisher noch nicht betrachteten Teil des Netzes nicht zu ignorieren. Ebenso werden die Aktivierungen, sowie die Fehlerwerte der aktuell betrachteten Schicht gebraucht. Ein Skalar $s$ zur Anpassung der Änderungen, welcher also festlegt, wie stark sich die Synapsengewichte nun verändern sollen, wird ebenfalls notwendig sein. Dieser Skalar wird auch Lernrate (des Netzes; eng.: *learning rate*) genannt. Nach dem Gradientverfahren wird versucht, die Fehlerwerte zu minimieren.

Stellt man sich das Netz als eine Art Funktion vor, so soll nun ein (meist lokaler) Tiefpunkt gefunden werden, an welchem die Fehler minimal sind.

---

[9] vgl. Patterson, Dan (1997): *Künstliche neuronale Netze: das Lehrbuch*. München: Prentice Hall. (S.162)

Um herauszufinden, in welche Richtung man mit welchem Maß vorgehen soll, um jenen Tiefpunkt zu erreichen, muss man erkennen, wie stark / schnell sich die Aktivierungen, in Anbetracht ihrer Fehlerwerte, verändern. Man braucht also die Steigungen jener Aktivierungen, welche durch das Ableiten erzeugt werden. Diese Kombination aus Fehlerwerten und jener Ableitung wird Gradient $g$ gennant. Es wird also nach dem Gradientverfahren wie folgt vorgegangen:

$$\Delta w = s * g_J * a_I^T$$

$$\text{wobei } g = e * \sigma'(a)$$

$$\text{mit } \sigma'(z) = \sigma(z) * (1 - \sigma(z))$$

Durch Einsetzen in die obige Gleichung erhält man:

$$\Delta w = s * e_J * (\sigma(a_J) * \left(1 - \sigma(a_J)\right)) * a_I^T$$

Da bei dem eigenen Netz die Aktivierungsfunktion schon bei Bestimmung der Aktivierung eines Neurons angewandt wird, kann sie hier weggelassen werden.

$$\Delta w = s * e_J * (a_J * (1 - a_J)) * a_I^T$$

Ebenfalls muss festgelegt werden, wie die Fehlerwerte einer Schicht ungleich der Outputschicht berechnet werden.

Es sollen also die bekannten Fehler der Schicht *J* auf Schicht *I* übertragen werden, in Anbetracht der einzelnen Synapsengewichte. Für den Fehler *e* eines Neurons $i$ in einer davorliegenden Schicht gilt also:

$$e_i = \sum_j \frac{w_{ji}}{\sum_j w_{ji}} * e_j$$

$w_{ji}$ ist hier das Gewicht $w$ der Synapse, die von Neuron *j* zu Neuron $i$ verläuft.

Da $\Delta w$ schon durch $s$ angepasst wird, kann die Normierung hier weggelassen werden:

$$e_i = \sum_j w_{ji} * e_j$$

### 2.2.5 Vorstellung der Bibliothek TensorFlow

TensorFlow ist eine Open Source Bibliothek von Google, die es ermöglicht, neuronale Netze nativ in Python zu generieren und zu trainieren. Hierbei werden verschiedene Voreinstellungsmöglichkeiten bereitgestellt. Zum aktuellen Zeitpunkt ist TensorFlow eine der beliebtesten Bibliotheken in der

Kategorie Machine Learning.

## 2.3 Anwendungsbeispiel Handschrifterkennung

Für den Trainings- und Evaluationsprozess der Netze werden in dieser Arbeit Bilder handschriftlich gezeichneter Ziffern verwendet. Die Pixel einer solchen Bilddatei können mit Farbwerten dargestellt werden. Für diese Arbeit werden nur Graustufenbilder verwendet. Jeder Pixel eines Bildes kann hier also durch eine ganze Zahl des Intervalls [0,255] dargestellt werden, wobei 0 weiß und 255 schwarz darstellt. Ein Bilddatensatz besteht also aus all jenen Werten einer einzelnen Bilddatei (hier: 784 Pixel pro Bild).

## 2.4 Erwartungen

Zur Evaluation des Netzes wird die Datenbank „The MNIST database of handwritten digits“[10] verwendet. Diese beinhaltet 60,000 handschriftliche Ziffern zum trainieren und 10,000 zum evaluieren (testen). In der Quelle der Datenbank sind durchschnittliche Genauigkeiten von über 95% aus verschiedenen Quellen angegeben.[11] Das eigene Netz wird wahrscheinlich eine etwas niedrigere Genauigkeit als jene erzielen. Hiermit wird vermutet, dass das eigene Netz eine Mindestgenauigkeit von 80% erreichen wird. Es wird im Idealfall auf eine Genauigkeit von ca. 90% geschätzt. Ebenso wird vermutet, dass das knN, welches mit TensorFlow erstellt wurde, in der Lage sein wird, eine Genauigkeit von bis zu 95% zu erzielen.

# 3 Praktischer Teil

## 3.1 Implementierung

Zur folgenden Implementierung ist ein Klassendiagramm im Anhang [Seite 17] beigelegt. Dieses Diagramm wurde für diese Arbeit entworfen und dient als Vorlage zur Implementierung. Zu Beginn werden die Gewichte der Synapsen, sowie die Bias der Neuronen zufällig gewählt. In der Implementierung wird ein

[10] siehe *http://yann.lecun.com/exdb/mnist/* ; hier verwendete Version heruntergeladen von: *https://www.python-course.eu/neural_network_mnist.php* (.csv-Format; in .pkl umgewandelt)

[11] siehe *http://yann.lecun.com/exdb/mnist/*

knN neuronales Netz genannt. Es wird in englischer Sprache programmiert, um dem Großteil der heutigen Literatur zu diesem Themengebiet nahe zu bleiben.
Für Matrizen wird die Bibliothek numpy, für grafische Darstellungen die Bibliothek matplotlib(.pyplot) verwendet.

### 3.1.1 Grundlegende Klassenstrukturen

Die grundlegende Implementierung der Klassen lässt sich im Anhang [Seite 18] wiederfinden. Die Netzwerkschichten (*layers*) sollen dabei so abgespeichert werden, dass eine Liste verschiedene Unterlisten mit den einzelnen Neuronen enthält. Jede Unterliste bildet dabei eine Schicht des knN.
Die Synapsen (*synapses*) werden aus Gründen, die in 3.1.2 deutlich werden, anders abgespeichert. Hier enthält eine Liste verschiedene Unterlisten, welche wiederum in weitere Listen aufgeteilt sind, die dann jeweils die Synapsen enthalten. Alle Synapsen, die zu einem Neuron einer Schicht, welche nicht der Inputschicht entspricht, führen, sind also in einer Liste aufgeführt. All jene Listen, welche Synapsen zwischen zwei Schichten beinhalten, werden wieder in einer Liste sortiert. Zuletzt werden alle Listen in eine einzelne Liste gespeichert. D.h. es findet eine Sortierung nach Schichten und nach Neuronen statt. Die Gewichte (*weights*) werden nach dem gleichen Prinzip gespeichert.

### 3.1.2 Feedforward-Algorithmus

Nach dem Prinzip aus 2.2.2 wird nun Feedforward implementiert.
Zuerst wird der Datensatz auf die Inputschicht geladen (siehe Anhang [Seite 18]).
Nun müssen die Aktivierungen jeder weiteren Schicht berechnet werden.
Für ein Neuron $j$ aus einer darauffolgenden Schicht gilt:

$$a_j = \sigma(z_j)$$

$$\text{wobei } z_j = \sum_i (a_i * w_{ji}) + b_j$$

$z$ ist die Summe aus allen Produkten bestehend aus den Synapsengewichten und den Inputs, zu welcher der Bias des betrachteten Neurons addiert wird.
Hierbei ist $a_i$ die Aktivierung $a$ eines Neurons $i$ in der vorhergehenden Schicht.
$w_{ij}$ ist das Gewicht $w$ der Synapse, die von Neuron $i$ zu Neuron $j$ verläuft.

$b_j$ ist das Bias $b$ des aktuell betrachteten Neurons $j$.

Eine Schleife, welche $a$ für jedes Neuron einer Schicht berechnet, liegt als erste Idee zur Implementierung nahe. Um während der Methode einfach auf die Synapsen, welche zu einem Neuron führen, zugreifen zu können, muss die Klasse Neuron erweitert werden. Jedes Neuron soll eine Liste besitzen, in der die angeschlossenen Synapsen gespeichert werden. Der Name „synapseNeighbours“ (Synapsennachbarn) liegt hier nahe. Natürlich muss in diesem Fall jene Liste im Vorfeld generiert werden, bevor die Berechnung der Aktivierungen stattfinden kann.

Ein erster Implementierungsversuch nach dieser Idee kann im Anhang [Seite 19] gefunden werden.

Mit dieser Funktion kann ein Datensatz erfolgreich durch das Netz geführt oder ge*feedet* werden. Dies klappt auch ganz gut. Führt man sich nun allerdings vor Augen, dass man auch Netzstrukturen mit Hunderten und Tausenden von Neuronen betrachten muss, so erscheint diese Vorgehensweise ungeeignet. Da jede Aktivierung von unterschiedlichen Synapsengewichten abhängt, muss jedes Neuron wohl einzeln betrachtet werden. Die Synapsengewichte können allerdings auch auf anderem Wege genutzt finden. Es wird also versucht, die Schleife über die einzelnen Synapsen zu ersetzen. Hierzu hilft es, sich die oben aufgeführte Gleichung erneut anzusehen.

$$z_j = \sum_i (a_i * w_{ji}) + b_j$$

kann auch wie folgt dargestellt werden:

$$z_j = a_{i_1} * w_{ji_1} + a_{i_2} * w_{ji_2} + \ldots + a_{i_n} * w_{ji_n} + b_j$$

*I* und *J* seien nun zwei aufeinanderfolgene Schichten, wobei *I* vor *J* liegt.

Man kann $z_j$ auch mit Hilfe von Matrizen darstellen:

$$z_j = \begin{bmatrix} w_{ji_1} \\ w_{ji_2} \\ \vdots \\ w_{ji_n} \end{bmatrix} \cdot \begin{bmatrix} a_{i_1} \\ a_{i_2} \\ \vdots \\ a_{i_n} \end{bmatrix} + b_j$$

Nun lassen sich die Matrizen auf alle Neuronen der Schicht erweitern:

$$\begin{bmatrix} z_{j_1} \\ z_{j_2} \\ \vdots \\ z_{j_n} \end{bmatrix} = \begin{bmatrix} w_{j_1 i_1} & w_{j_1 i_2} & \cdots & w_{j_1 i_n} \\ w_{j_2 i_1} & w_{j_2 i_2} & \cdots & w_{j_2 i_n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{j_m i_1} & w_{j_m i_2} & \cdots & w_{j_m i_n} \end{bmatrix} \times \begin{bmatrix} a_{i_1} \\ a_{i_2} \\ \vdots \\ a_{i_n} \end{bmatrix} + \begin{bmatrix} b_{j_1} \\ b_{j_2} \\ \vdots \\ b_{j_4} \end{bmatrix}$$

Nun lassen sich folgende Gleichungen aufstellen:

$$z_j = a_I * w_{jI} + b_j$$

$$z_J = a_I * w_{JI} + b_J$$

Hierbei ist $a_I$ eine $n$-zeilige, einspaltige Matrix, welche alle Aktivierungen der Schicht $I$ enthält. $w_{IJ}$ ist eine Matrix mit $n$ Zeilen und $m$ Spalten, die alle Gewichte der Synapsen beinhaltet, welche zwischen Schicht *I* und *J* liegen.
Mit dieser Umformung wird die Methode nun umgeschrieben (siehe Anhang [Seite 19]).
Man beachte die Tatsache, dass *synapseNeighbours* nun nicht mehr gebraucht wird. Ebenso werden nun die Aktivierungen der vorhergehenden Schicht mithilfe einer Schleife in *inputMatrix* abgespeichert.
Zuletzt werden die Aktivierungen der letzten Schicht zurückgegeben.

### 3.1.3 Backpropagation-Algorithmus

Um das Netz nun verbessern zu können, wird Backpropagation nach dem Prinzip aus 2.2.3 implementiert. Die Fehlerberechnung der Outputschicht soll wie folgt stattfinden:

$$e_{output} = a_{output} - a_d$$

Hierbei ist $a_d$ die Matrix $a$ mit den erwünschten Outputaktivierungen zu einem zuvor durch das Netz gefeedeten Datensatz $d$. Da nun von der letzten Schicht ausgehend vorgegangen wird, muss die Matrix mit den Synapsengewichten, welche in 3.1.2 genutzt wurde, transponiert werden (Achseninvertierung). Transponierte Matrizen werden mit einem hochgestellten *T* gekennzeichnet (Implementierung ist im Anhang auf [Seite 19] vorzufinden).
Es werden einige Hilfsvariablen initialisiert und für jede weitere Schicht die aktuellen Gewichte und Synapsen aktualisiert. Um später mit den Fehlerwerten rechnen zu können, müssen sie in eine einspaltige Matrix (Vektor) umgeformt werden. Die Werte der aktuell betrachteten Schicht (Outputs) und der Schicht davor (Inputs) werden gespeichert. Die Inputs müssen wieder transponiert werden. Die Fehlermatrix muss für weitere Berechnungen als einzeilige Matrix vorliegen (siehe Anhang [Seite 20]). Die Berechnung der Änderung der Synapsengewichte wird nach dem Prinzip aus 2.2.4 implementiert:

```
deltaWeights=self.learningrate * errorMatrix * (outputMatrix*(1-outputMatrix)) * transposedInputMatrix
```

Zuletzt werden die aktuellen Synapsengewichte angepasst. Außerdem müssen die Fehler der als nächstes zu betrachtenden Schicht berechnet werden. Dies passiert ebenfalls nach dem in 2.2.4 erläuterten Ablauf. Es muss zusätzlich eine Methode implementiert werden, um die Matrix zu aktualisieren, welche alle Synapsengewichte enthält (hier: updateSynapseWeightMatrix; siehe Anhang [Seite 20]).

## 3.2 Analyse

Für das effiziente Training müssen Methoden implementiert werden, die das Trainieren mithilfe von Schleifen ermöglichen, sodass nicht mehr jeder Datensatz manuell eingegeben werden muss, sondern nur noch ein Intervall der zu trainierenden Datenmenge. Ebenso werden Funktionen erzeugt, welche den Verlauf der Genauigkeit nach dem Training grafisch darstellen lassen und welche das Speichern sowie Laden des Netzwerkes möglich machen.
Es wird ein Netz A mit vier Schichten erstellt. Dabei hat die Inputschicht 784 und die Outputschicht zehn Neuronen. Die verborgenen Schichten haben alle den gleichen Abstand zu den jeweils benachbarten Schichten (Schichtengrößen: [784, 526, 268, 10]). Das Netz wird zunächst auf den 60,000 Trainingsdatensätzen trainiert. Daraufhin wird eine Voraussage für jedes Bild der Testdatenbank getroffen und mit den Labels verglichen. Es wird ebenfalls ein knN mit TensorFlow erstellt und trainiert, welches dieselbe Struktur aufweist. Hier werden allerdings andere Parameter für das Training gewählt (siehe 5.2). Das TF-knN wird in einem Jupyter Notebook erstellt (siehe Anhang [Seite 21]).

### 3.2.1 Verschiedene Netzstrukturen

Es findet ebenfalls ein Vergleich verschiedener Netzstrukturen statt. Hierbei werden nur die ersten 10000 Datensätze zum Trainieren verwendet. Es werden verschiedene Aspekte, wie beispielsweise die benötigte Zeit pro Datensatz (bei Backpropagation) oder die Endgenauigkeit untersucht. Zu diesem Zweck sollen fünf weitere Netze erstellt werden. Es sollen einerseits Netze untersucht werden, bei denen die verborgenen Schichten die gleiche Anzahl Neuronen haben wie die Input- oder Outputschicht. Netz B hat also [784,10,10,10] Neuronen, Netz C [784,784,784,10] Neuronen. Ein Netz, in welchem die

Anzahl der Neuronen mit jeder Schicht wächst, soll ebenfalls betrachtet werden. Netz D besteht hier aus [784,892,1000,10] Neuronen. Ein Netz mit einem Aufbau ähnlich dem von Netz A, allerdings mit nur einer verborgenen Schicht soll ebenfalls evaluiert werden. Netz E hat [784,397,10] Neuronen. Zuletzt soll ein Netz ohne verborgene Schichten inspiziert werden. Das Netz F hat somit [784,10] Neuronen. Jede Netzstruktur wird mit den Lernraten 1, 0,1 und 0,01 getestet. Auch hier sollen wieder Netze mit den jeweiligen Strukturen in TensorFlow erstellt, trainiert und verglichen werden. Die verschiedenen Genauigkeitskurven der eigenen Netze sind samt Messwerten im Anhang [Seite 21f.] vorzufinden. Die Kurven sind hierbei in Abhängigkeit der Lernrate jeweils blau (1), orange (0,1) und grün (0,01) gefärbt.

### 3.3 Verarbeitung selbstständig erstellter Daten (/Zeichnungen)

Zu diesem Zeitpunkt sind die sechs Netzwerke trainiert und nun können Zusammenhänge und Differenzen aus den erhobenen Daten erschlossen werden. Vorher soll hier ein solches knN ein wenig greifbarer gemacht werden. Zu diesem Zweck soll eine minimalistische GUI verwendet werden, die das Zeichnen eigener Ziffern ermöglicht. Hierfür wird eine bereits existierende Oberfläche aus einem Internetforum modifiziert. Die Quelle, sowie der modifizierte Quellcode sind im Anhang [Seite 24] angegeben.
Diese Zeichnungen sollen dann als Bilddateien gespeichert und durch das vollständig trainierte Netz A gefeedet werden, woraufhin das knN eine Voraussage für die Zeichnungen anfertigt. Hiermit ist es möglich, das Netz zu nutzen, um eigene Zeichnungen evaluieren zu können. Hierzu sind Bildschirmaufnahmen im Anhang [Seite 23] beigelegt.

## 4 Ergebnisse

### 4.1 Rechnereigenschaften

In diesem Teil der Arbeit werden konkrete Zeitwerte genannt. Alle Messungen wurden auf dem gleichen Rechner unternommen. Es wird nur mithilfe des Prozessors, nicht mithilfe der Grafikkarte trainiert/evaluiert. Die genauen Spezifikationen des Rechners sind im Anhang [Seite 24] vorzufinden.

## 4.2 Genauigkeitsverlauf

Auf den 10,000 Testdaten erreicht das knN A (siehe 3.2) eine Trefferwahrscheinlichkeit von ca. 95,16%. Der Genauigkeitsverlauf des Netzes während dem Trainieren auf den Trainingsdaten ist rechts abgebildet. Es lässt sich eine massive Verbesserung der Genauigkeit im Bereich bis ca. 5000 verwendeter Datensätze feststellen. Danach verläuft die Genauigkeitskurve asymptotisch zum Wert 1 (100%). Das Netz benötigt hierbei ca. eine Sekunde pro Datensatz. D.h. für einen vollständigen Durchlauf mit 60,000 Datensätzen wurden knapp 17 Stunden Zeit benötigt.

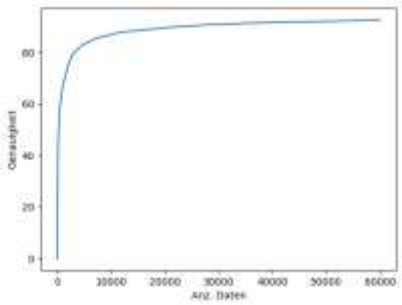



*Abbildung 3: Netz A: Genauigkeitskurve*

## 4.3 Vergleich mit TensorFlow

Es wird ein knN mit dem Modul TensorFlow generiert, welches die gleiche Netzstruktur wie Netz A besitzt (für Implementierung siehe Anhang [Seite 21]). Es wird auf den gleichen Datensätzen wie das eigene Netz trainiert. Es erreicht eine Genauigkeit von ca. 96,59% auf den Testdaten. Die Netzstrukturen aus 4.2 werden ebenfalls mit TensorFlow getestet. Die Genauigkeiten für Strukturen B, C, D, E und F betragen jeweils 49,18%; 89,06%; 88,9%; 87,2% und 76,97% (auf 10,000 Trainingsdaten).

# 5 Diskussion

## 5.1 Differenzen zwischen den Strukturen

Durch die Analyse werden mehrere Erkenntnisse feststellbar. Zum einen benötigen Netze mit einer höheren Anzahl an Neuronen auch mehr Zeit zum Verarbeiten von Daten / Optimieren des Netzes. Ebenfalls stellt man fest, dass bei geringer Lernrate Netze mit einer höheren Anzahl Neuronen besser lernen als Netze mit einer geringeren Anzahl Neuronen. Dies lässt sich dadurch erklären, dass bei einem Netz mit einer höheren Anzahl an Neuronen jede einzelne Synapse weniger stark angepasst werden muss, um ein optimales

Resultat erzielen zu können, da allgemein mehr Synapsen vorhanden sind. Davon abgesehen gibt es einen weiteren wichtigen Grund, warum zu hohe Lernraten sich negativ auf die Genauigkeit eines knN auswirken können.

Stellt man sich das Netz wieder als Funktion vor, bei welcher ein Minimum (der Fehler) gesucht wird, so kann es bei zu hohen Lernraten vorkommen, dass das Minimum übersprungen wird. Hier spricht man von *„overshooting“*. Dieser Fall ist in der Grafik auf der rechten Seite (stark vereinfacht) zu erkennen.

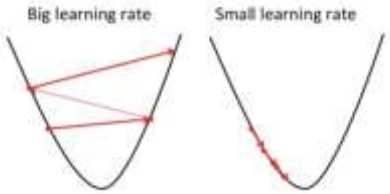



*Abbildung 4: Differenz zwischen hoher und geringer Lernrate (https://towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b ; 26.03.2019)*

Bei geringer Lernrate nähert sich die Genauigkeit langsam, aber sicher an ein lokales oder globales Minimum an, während hingegen bei hoher Lernrate das Minimum übersprungen wird und die Genauigkeit sich nicht dem erwünschten Wert nähert. Jedes Netz hat also eine ideale Lernrate, mit welcher es am effizientesten arbeiten kann. Bei den hier betrachteten Netzen lag diese immer im Intervall [0,1;0,01]. Ebenso muss die Anzahl der Neuronen eines Netzes passend zum zu lösenden Problem gewählt werden.

Zu wenige Neuronen sorgen dafür, dass das Netz nicht komplex genug ist und die maximal erzielbare Genauigkeit zu tief liegt. Zu viele Neuronen sorgen hingegen für viel zu hohe Zeitkosten, erhöhen die maximale Genauigkeit des Netzes aber nur minimal, wenn überhaupt. Für einen genaueren Effizienzvergleich der verschiedenen Netzstrukturen wird ein Quotient $q$ wie folgt definiert:

$$q_n = \frac{g_n}{l_n}$$

Wobei $g$ die erzielte Höchstgenauigkeit $(0 < g < 100)$ des Netzes $n$ und $l$ die Anzahl der Neuronen des Netzes darstellt.

Nun werden die verschiedenen Werte in einer Tabelle dargestellt:

*Tabelle 1: Genauigkeitsvergleich verschiedener Netzstrukturen*

| | Netz B | Netz C | Netz D | Netz E | Netz F |
|---|---|---|---|---|---|
| g | 58,92 | 86,55 | 86,81 | 87,56 | 49,85 |
| l | 814 | 2362 | 2686 | 1191 | 794 |
| q | 0,0724 | 0,0366 | 0,0322 | 0,0735 | 0,0628 |

Man merkt schnell, dass es (für dieses Problem) eine ideale Netzstruktur geben muss, bei welcher mit der geringsten Anzahl Neuronen die höchste Genauigkeit erzielt wird. Beispielsweise wird aus den Daten erkenntlich, dass eine verborgene Schicht, welche mehr Neuronen als die Inputschicht besitzt, nicht effizient ist. Hier wurde der Zeitaufwand nicht direkt betrachtet.
Ersetzt man hingegen die Anzahl der Neuronen $l$ durch die benötigte Zeit $t$, so wird man feststellen, dass Netze mit möglichst geringen Zeitkosten immer einen höheren Quotienten erhalten werden als komplexere, präzisere Netze. Dies liegt daran, dass die hier erstellte Implementierung im Vergleich zu anderen deutlich langsamer arbeitet (siehe 5.2).

## 5.2 Differenzen zwischen eigenem Netz und TensorFlow-Netz

Der Quotient $q$ wird auch hier wieder für jedes Netz berechnet. Hier soll nun aber zusätzlich die insgesamt benötigte Zeit $t$ betrachtet, um die jeweiligen Genauigkeiten zu erreichen. Der Quotient $p$ mit

$$p_n = \frac{t_n}{l_n}$$

soll die benötigte Zeit pro Neuron des Netzes darstellen. Die eigene Implementierung wird mit TensorFlow verglichen, letztere ist hierbei orange markiert.

*Tabelle 2: Effizienzvergleich zwischen eigener Implementierung und TensorFlow*

| | B | B | C | C | D | D | E | E | F | F |
|---|---|---|---|---|---|---|---|---|---|---|
| $g$ | 58,92 | 49,18 | 86,55 | 89.06 | 86,81 | 88,90 | 87,56 | 87,20 | 49,85 | 76,97 |
| $l$ | 814 | 814 | 2362 | 2362 | 2686 | 2686 | 1191 | 1191 | 794 | 794 |
| $q$ | 0,0724 | 0.072 | 0,0366 | 0.037 | 0,0322 | 0.032 | 0,0735 | 0.074 | 0,0628 | 0,063 |
| $t$ | 4,3min | 1s | 7,5h | 10s | 9,83h | 12s | 1,97h | 3s | 3,93min | 1s |
| $p$ | 3,13 | 0.001 | 0,087 | 0.004 | 0,076 | 0.004 | 1,68 | 0.003 | 3.364 | 0.001 |

Es stellt sich heraus, dass die auf TensorFlow basierenden Netze in den meisten Fällen eine ungefähr gleich hohe maximale Genauigkeit erzielen wie die eigene Implementierung. Allerdings wird auch deutlich, dass die TensorFlow-Netzwerke um ein Vielfaches schneller arbeiten als die eigene Implementierung. Die Tensorflow-Netze unterscheiden sich zwar in mehreren Faktoren (z.B. Aktivierungsfunktion oder Optimierungsverfahren) von der

eigenen Implementierung, allerdings sollten Netze mit gleicher Struktur und ausreichend Training die gleiche Höchstgenauigkeit erreichen. Es wird also vermutet, dass die großen Geschwindigkeitsdifferenzen zwischen der eigenen Ausarbeitung und TensorFlow auf die konkreten Implementierungen zurückzuführen ist.

## 5.3 Fazit

Es lässt sich feststellen, dass es für jedes Problem eine optimale Netzstruktur geben muss. Moderne Netzwerke unterscheiden sich von traditionellen Modellen vor allem in der Datenverarbeitungsgeschwindigkeit. Das Prinzip blieb also größtenteils erhalten, die Trainingsmethoden haben sich allerdings deutlich verbessert. Die zu Beginn aufgestellten Vermutungen lassen sich bestätigen, die eigene Implementierung hat jene Erwartungen moderat übertroffen.

# 6 Ausblick

Durch diese Arbeit wurde ein erster Einblick in die Welt der künstlichen neuronalen Netze ermöglicht. Eine interessante Fortsetzung wäre hier auf jeden Fall die Betrachtung einer eigenen Implementierung mit den Eigenschaften und Optimierungsmaßnahmen des hier analysierten TensorFlow-Netzes. So könnte man die konkreten Implementierungen unabhängig von den gewählten Optimierungsmaßnahmen untersuchen. Ebenfalls wurden die Trainingsdaten hier immer nur einmalig von den Netzen durchlaufen. Es bietet sich also an, zu untersuchen ob und wie stark sich die Genauigkeiten bei mehrfachem Durchlaufen ändern würden, und wie stark die Netze an Überanpassung (eng.: *„overfitting“*) leiden würden. Dennoch wurde in dieser Arbeit lediglich ein winziger Teil der gigantischen Welt des maschinellen Lernens betrachtet. Neben den hier analysierten klassischen Feedforward-Netzwerken sind heute beispielsweise auch rekurrente (eng.: „*recurrent“*) oder „gefaltete“ (eng.: „*convolutional“*) Netze vorzufinden. Neben TensorFlow exisiteren natürlich auch viele weitere Bibliotheken mit ähnlichen Zwecken und Funktionsweisen. Die durch diese Arbeit errungenen Erkenntnisse und Einblicke sollen Interesse an diesem Themengebiet der Informatik wecken, wenn nicht sogar verstärken.

# Anhang

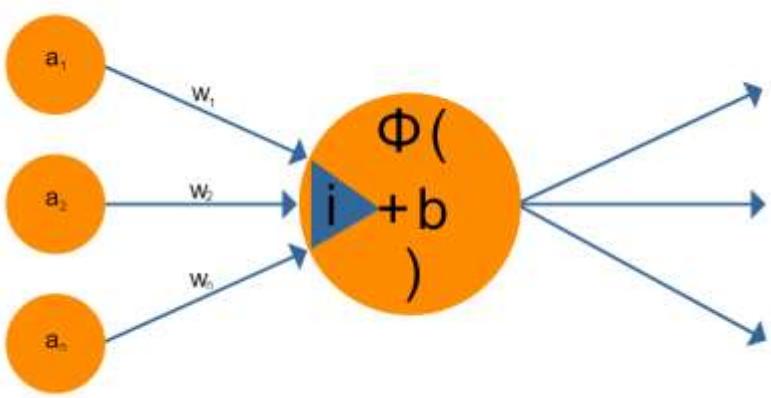



*Abbildung 5: Künstliches Neuron*

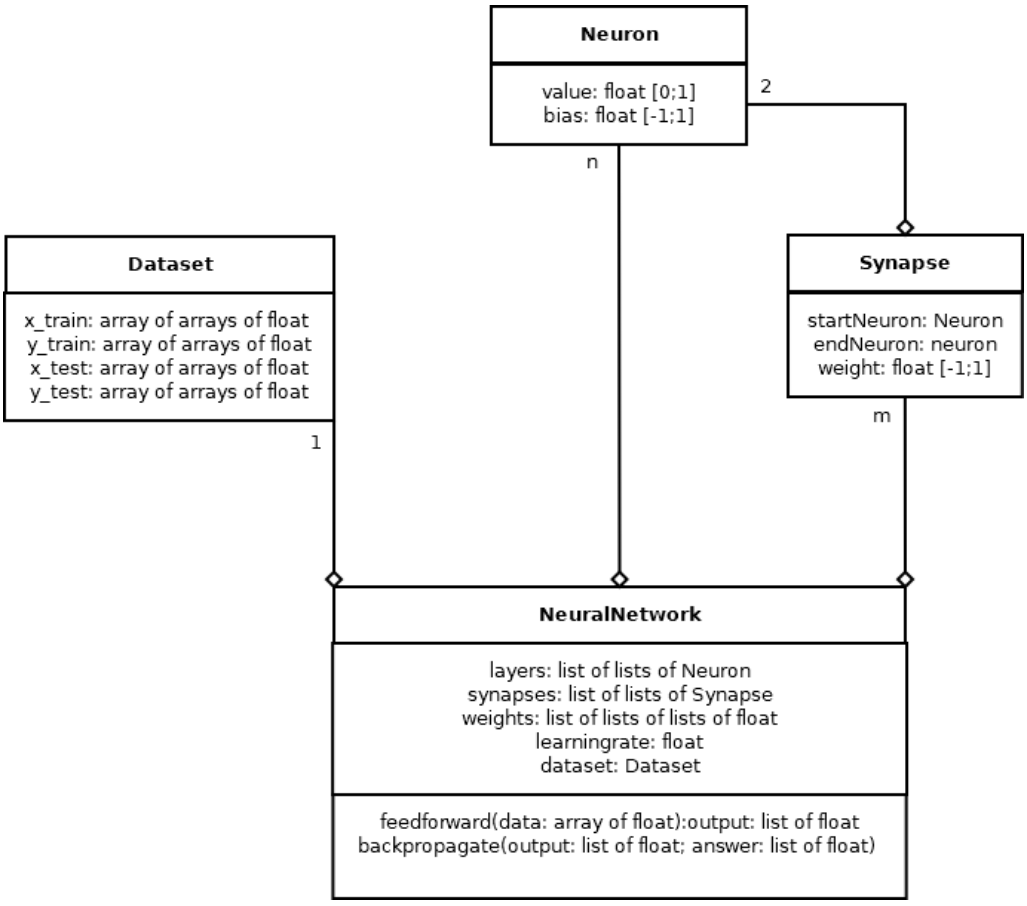



*Abbildung 6: Klassendiagramm knN*

```
class Dataset(object):
    """
    x_train: Trainingsdaten: array of arrays of float
    y_train: Trainingslabels: array of arrays of float
    x_test: Testdaten: array of arrays of float
    y_test: Testlabels: array of arrays of float
    """
    def __init__(self,dataset,x_train=None,x_test=None,y_train=None,y_test=None):
        self.dataset=dataset
        self.x_train=x_train
        self.x_test=x_test
        self.y_train=y_train
        self.y_test=y_test

class Neuron(object):
    """
    value,bias: float [0;1], float [-1;1]
    """
    def __init__(self,value=1):
        self.value=value
        self.bias=np.random.uniform(-1,1)

class Synapse(object):
    """
    startNeuron, endNeuron: Neuron
    weight: float [-1,1]
    """
    def __init__(self,startNeuron,endNeuron):
        self.startNeuron=startNeuron
        self.endNeuron=endNeuron
        self.weight=np.random.uniform(-1,1)

class NeuralNetwork(object):
    """
    layers: list of lists of Neuron
    synapses: list of lists of lists of Synapse
    weights: list of lists of lists of float
    learningrate: float
    dataset: Dataset
    """
    def __init__(self,dataset):
        self.layers=[]
        self.synapses=[]
        self.weights=[]
        self.learningrate=0
        self.dataset=dataset
```

*Abbildung 7: Grundlegende Implementierung der Klassen*

```
def feedforward(self,data):
    """
    input: data: array of int (0<x<1)
    output: outputValues: array of int(0<x<1)
    """
    #initialise input-layer
    for neuron in self.layers[0]:
        index = self.layers[0].index(neuron)
        neuron.value=data[index]
```

*Abbildung 8: Feedforward: Initialisierung*

```
#feedforward
layerIndex=0
while layerIndex < len(self.layers):

    #calculate every activation for neurons in next layer
    for neuron in self.layers[layerIndex]:
        newValue = 0
        nextNeuron = neuron.synapseNeighbours[0].endNeuron
        nextNeuronBias = nextNeuron.bias

        for synapse in neuron.synapseNeighbours:
            startNeuronValue = synapse.startNeuron.value

            synapseWeight = synapse.weight
            newValue += (startNeuronValue*synapseWeight) / len(neuron.synapseNeighbours)

        newValue += nextNeuronBias
        nextNeuron.value = sigmoid(newValue)

    layerIndex += 1
```

*Abbildung 9: Feedforward: erster Ansatz*

```
#feedforward
layerIndex = 1

while layerIndex < len(self.layers):
    inputMatrix = []

    for neuron in self.layers[layerIndex-1]:
        inputMatrix.append(neuron.value)
    neuronIndex = 0

    for neuron in self.layers[layerIndex]:
        endNeuronBias = neuron.bias

        weightMatrixRow = self.weights[layerIndex-1][neuronIndex]
        newValue = np.matmul(weightMatrixRow,inputMatrix)

        neuron.value = sigmoid(newValue + endNeuronBias)
        neuronIndex += 1

    layerIndex += 1
```

*Abbildung 10: Feedforward: Verbesserung*

```
def backpropagate(self,output,answer):
    """
    output,answer: arrays of float, same lenght
    """
    #begins at output
    #calculate output error
    outputMatrix = output
    answerMatrix = answer
    errorMatrix = outputMatrix - answerMatrix
```

*Abbildung 11: Backpropagation: Initialisierung*

```
#then goes from second last layer to first layer
for layer in self.layers[:-1]:
    layerCounter += 1
    layerIndex = len(self.layers) -2 - layerCounter

    currentSynapseMatrix = synapses[layerIndex]
    currentWeightMatrix = synapseWeights[layerIndex]

    previousErrorMatrix = np.copy(errorMatrix)
    previousErrorMatrix = np.reshape(previousErrorMatrix,(len(previousErrorMatrix),1))
    transposedWeightMatrix = np.transpose(currentWeightMatrix)

    outputMatrix = []
    for neuron in self.layers[layerIndex+1]:
        outputMatrix.append(neuron.value)
    outputMatrix=np.array(outputMatrix)

    inputMatrix = []
    for neuron in self.layers[layerIndex]:
        inputMatrix.append(neuron.value)
    inputMatrix = np.array(inputMatrix)

    transposedInputMatrix = np.reshape(inputMatrix,(len(inputMatrix),1))

    errorMatrix = errorMatrix.flatten()
```

*Abbildung 12: Backpropagation: Beginn der Schleife*

```
    #apply changes
    transposedCurrentSynapseMatrix = np.transpose(currentSynapseMatrix)
    arrayIndex = -1

    for changeValueArray in deltaWeights:
        arrayIndex += 1
        valueIndex =- 1

        for changeValue in changeValueArray:
            valueIndex+=1
            currentSynapse = transposedCurrentSynapseMatrix[arrayIndex][valueIndex]
            currentChangeValue = deltaWeights[arrayIndex][valueIndex]

            currentSynapse.weight -= currentChangeValue

    errorMatrix = np.matmul(transposedWeightMatrix,previousErrorMatrix)

self.updateSynapseWeightMatrix()
```

*Abbildung 13: Backpropagation: Aktualisieren der Synapsengewichte*

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

```
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)
anzahlEinträge=60000
x_train = mnist.train.images[:anzahlEinträge,:]
y_train = mnist.train.labels[:anzahlEinträge,:]

x_train=x_train[:10000]
y_train=y_train[:10000]

x_test = mnist.test.images[:anzahlEinträge,:]
y_test = mnist.test.labels[:anzahlEinträge,:]
```

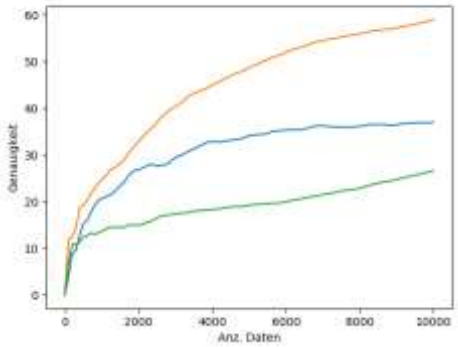
```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(526,tf.nn.relu))
model.add(tf.keras.layers.Dense(268,tf.nn.relu))
model.add(tf.keras.layers.Dense(10,tf.nn.softmax))

model.compile(optimizer="adam",
                    loss="categorical_crossentropy",
                    metrics=["accuracy"])
```

```
model.fit(x_train,y_train,epochs=1)
```

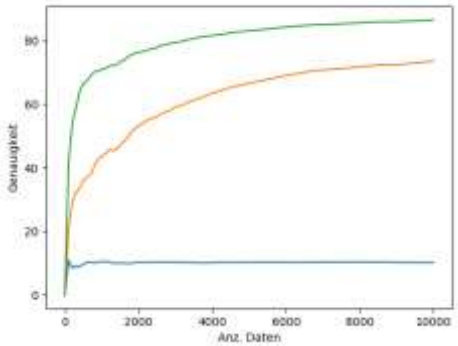
*Abbildung 14: knN mit TensorFlow*

Genauigkeitskurven:



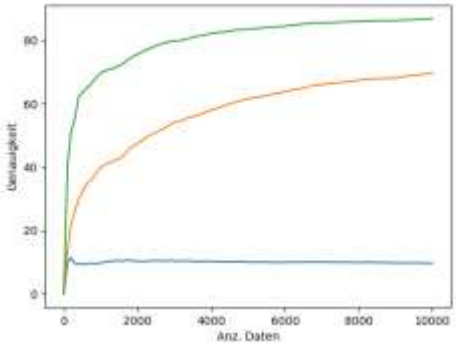

Schichtengrößen: [784,10,10,10]
Anzahl Neuronen: 814
Zeitaufwand pro Datensatz (Backpropagation): 0.026 s
Endgenauigkeit ($s = 1$): 37,04%
Endgenauigkeit ($s = 0{,}1$): 58,92%
Endgenauigkeit ($s = 0{,}01$): 26,62%

*Abbildung 15: Netz B: Genauigkeitskurven*



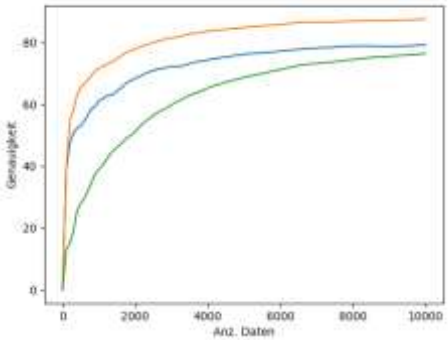

Schichtengrößen: [784,784,784,10]
Anzahl Neuronen: 2362
Zeitaufwand pro Datensatz (Backpropagation): 2.7 s
Endgenauigkeit ($s = 1$): 10,18%
Endgenauigkeit ($s = 0{,}1$): 73,59%
Endgenauigkeit ($s = 0{,}01$): 86,55%

*Abbildung 16: Netz C: Genauigkeitskurven*



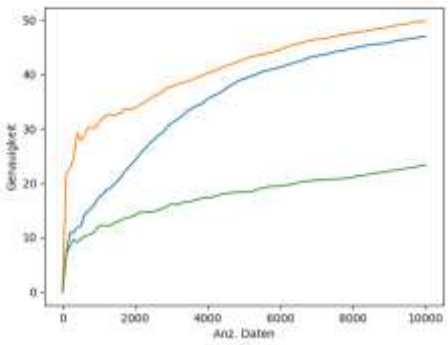

Schichtengrößen: [784,892,1000,10]
Anzahl Neuronen: 2686
Zeitaufwand pro Datensatz (Backpropagation): 3.54 s
Endgenauigkeit ($s = 1$): 9,73%
Endgenauigkeit ($s = 0{,}1$): 69,7%
Endgenauigkeit ($s = 0{,}01$): 86,81%

*Abbildung 17: Netz D: Genauigkeitskurven*




Schichtengrößen: [784,397,10]
Anzahl Neuronen: 1191
Zeitaufwand pro Datensatz (Backpropagation): 0.71 s
Endgenauigkeit ($s = 1$): 79,24%
Endgenauigkeit ($s = 0{,}1$): 87,56%
Endgenauigkeit ($s = 0{,}01$): 76,4%

*Abbildung 18: Netz E: Genauigkeitskurven*




Schichtengrößen: [784,10]
Anzahl Neuronen: 794
Zeitaufwand pro Datensatz (Backpropagation): 0.023 s
Endgenauigkeit ($s = 1$): 46,97%
Endgenauigkeit ($s = 0{,}1$): 49,85%
Endgenauigkeit ($s = 0{,}01$): 23,36%

*Abbildung 19: Netz F: Genauigkeitskurven*

Verarbeitung selbstständig erstellter Daten (/Zeichnungen):


*Abbildung 21: leeres Zeichenfenster*


*Abbildung 20: selbstständig gezeichnete Zwei*

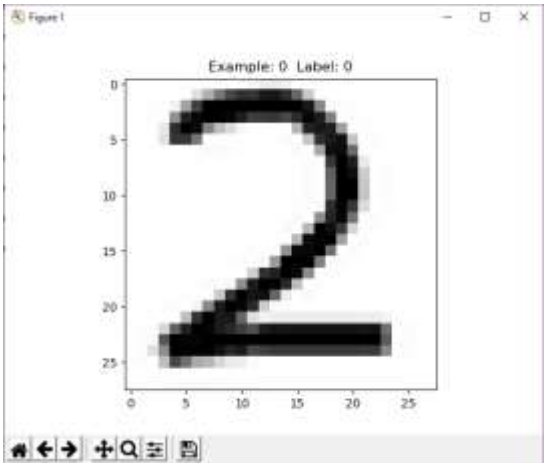



*Abbildung 22: abgespeicherte Datei der selbstständig gezeichneten Zwei (vergrößert)*

Voraussage des Netzes:

```
>>> nn.ownImage()
2
```

Modifizierter Quellcode (die ursprüngliche Version ist hier zu finden: https://stackoverflow.com/questions/52146562/python-tkinter-paint-how-to-paint-smoothly-and-save-images-with-a-different):

```
from tkinter import *
import PIL
from PIL import Image, ImageDraw
import numpy as np
import matplotlib.pyplot as plt

def save():
    global image_number
    filename = f'image_{image_number}.png'
    image1.resize((28,28), Image.ANTIALIAS).save(filename)
    pix_val=getData()
    npix_val=normaliseData(pix_val)
    npix_val=np.array(npix_val)
    np.save("OwnImageData", npix_val)


def activate_paint(e):
    global lastx, lasty
    cv.bind('<B1-Motion>', paint)
    lastx, lasty = e.x, e.y


def paint(e):
    global lastx, lasty
    x, y = e.x, e.y
    cv.create_line((lastx, lasty, x, y), width=28)
    draw.line((lastx, lasty, x, y), fill='black', width=28)
    lastx, lasty = x, y

def drawImage():
    global cv,root,lastx,lasty,draw,image_number,image1
    root = Tk()

    lastx, lasty = None, None
    image_number = 0

    cv = Canvas(root, width=280, height=280, bg='white')
    image1 = PIL.Image.new('RGB', (280, 280), 'white')
    draw = ImageDraw.Draw(image1)

    cv.bind('<1>', activate_paint)
    cv.pack(expand=YES, fill=BOTH)

    btn_save = Button(text="save", command=save)
    btn_save.pack()

    root.mainloop()
```

```
def getData(path="image_0.png"):
    im = Image.open(path)
    pix_val=list(im.getdata())

    return pix_val

def normaliseData(data):
    xmin=255
    xmax=0
    singleValues=[]
    normalisedData=[]
    for valueTuple in data:
        singleValue=np.amax(valueTuple)
        singleValues.append(singleValue)

    for value in singleValues:
        normalisedValue=(value-xmin) / (xmax-xmin)
        normalisedData.append(normalisedValue)

    return normalisedData

def showImage():
    a=getData()
    b=normaliseData(a)
    b=np.array(b)
    label=0
    image=b.reshape([28,28])
    plt.imshow(image, cmap=plt.get_cmap("gray_r"))
    plt.title('Example: %d  Label: %d' % (0, label))
    plt.show()
```

*Abbildung 23: modifizierter Quellcode der GUI*

Spezifikationen des benutzten Rechners für konkrete Zeitmessungen:
- Prozessor: Intel(R) Core(TM) i5-2320 CPU @ 3.00GHz  (Quad-Core)
- Arbeitsspeicher: 8,00 GB
- Systemtyp: 64-Bit-Betriebssystem, x64-basierter Prozessor

# Literaturverzeichnis

Simon Haykin (2009). *Neural Networks and Learning Machines*. Dritte Edition. Hamilton, Ontario, Canada: Pearson.

Oliver Kramer (2009). *Computational Intelligence.* Dortmund. Springer.

Dan W. Patterson (1997). *Künstliche neuronale Netze : das Lehrbuch*. München: Prentice Hall.

Daniel Shiffman. (2012) *The Nature Of Code* (https://natureofcode.com/book/):

# Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass ich die von mir eingereichte Facharbeit bzw. die von mir namentlich gekennzeichneten Teile selbständig verfasst und ausschließlich die angegebenen Hilfsmittel benutzt habe.

Kaiserslautern, den 03.04.2019

_____

(Name des Verfassers)

# Einverständniserklärung

Ich erkläre mich einverstanden/~~nicht einverstanden~~, dass meine Diplomarbeit an Personen, die nicht mittelbar oder unmittelbar an meiner Prüfung beteiligt sind, ausgeliehen wird.

Kaiserslautern, den 03.04.2019

_____

(Name des Verfassers)