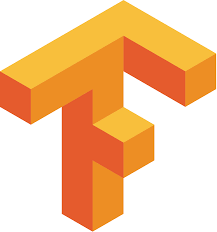
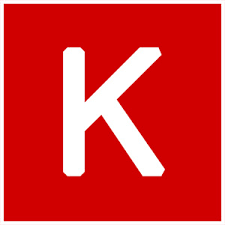
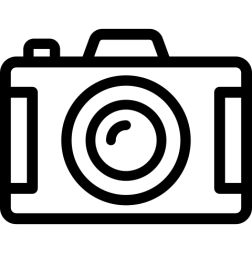
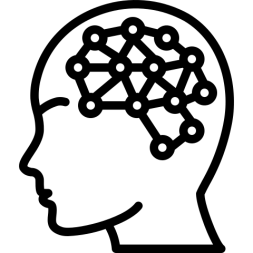
**Kann ein Computer lesen lernen?**

**Auslesen eines Nummernschildes über ein neuronales Netzwerks unter Benutzung von Tensorflow und Keras.**

****

****

**Fach: Informatik**

**Schule: Anno Gymnasium Siegburg**

**Verfasser: Philip Wentzke**

**Kurs: Grundkurs Informatik**

**Betreuer: Herr Tim Schmitz-Porten**

**Abgabetermin: 08.03.2019**

**Stufe: Q1**

**Schuljahr: 2018/2019**

**Inhaltsverzeichnis**

[1 Einleitung 3](#_Toc2458121)

[2 Was ist ein NN? 3](#_Toc2458122)

[3 Allgemeine Information zu NN 4](#_Toc2458123)

[3.1 Aufbau eines NNs 4](#_Toc2458124)

[3.1.1 Layer 5](#_Toc2458125)

[3.1.2 Neuron 5](#_Toc2458126)

[3.1.3 Connection 5](#_Toc2458127)

[3.1.4 Bias 5](#_Toc2458128)

[3.1.5 Weight 5](#_Toc2458129)

[3.1.6 Aktivierungsfunktion 5](#_Toc2458130)

[3.1.7 Konventionen zum Aufbau eines NNs 6](#_Toc2458131)

[3.2 Benutzung 7](#_Toc2458132)

[3.2.1 Trainingsfunktionen 7](#_Toc2458133)

[3.2.2 Backpropagation 7](#_Toc2458134)

[3.2.3 Anwendungsbeispiele 8](#_Toc2458135)

[4 Was sind Tensorflow und Keras? 8](#_Toc2458136)

[4.1 Erklärung von Tensorflow und Keras 8](#_Toc2458137)

[4.2 Wie wird mit Tensorflow ein NN modelliert? 9](#_Toc2458138)

[5 Vorbereitung der Daten 9](#_Toc2458139)

[5.1 Trainingsdaten-Set 9](#_Toc2458140)

[5.2 Testdaten 9](#_Toc2458141)

[6 Aufbau und Anbindung eines NNs zum Auslesen eines Nummernschildes 11](#_Toc2458142)

[6.1 Pythoncode 11](#_Toc2458143)

[6.2 Javacode 11](#_Toc2458144)

[7 Evaluation der Ausgabe des Programmes 12](#_Toc2458145)

[8 Fazit 13](#_Toc2458146)

[Literaturverzeichnis 14](#_Toc2458147)

[Bildquellen 14](#_Toc2458148)

[Textquellen 14](#_Toc2458149)

[Abbildungsverzeichnis 15](#_Toc2458150)

[Tabellenverzeichnis 15](#_Toc2458151)

[Formelverzeichnis 15](#_Toc2458152)

[Anhang xvi](#_Toc2458153)

# Einleitung

Ich habe mir für meine Facharbeit die Frage gestellt, ob Computer lesen lernen können. Dies bezieht sich auf das, speziell in letzter Zeit, Aufkommen des Themas künstliche Intelligenz. Wenn man einem Computer das Auslesen von Texten auf vorgegebenen Bildern beibringen könnte, kann man sich vorstellen, dass Computer bald Menschen noch ähnlicher werden könnten und somit näher an der künstliche Intelligenzen kommen könnten.

Ich versuche in dieser Arbeit, als Beispiel für das Lesen von Bilder, einem Computer das Auslesen eines Nummernschildes beizubringen. Dafür benutze ich eine Methode, die in der aktuellen Forschung zum Thema künstlicher Intelligenzen am nächsten an eine sogenannte KI herankommt. Diese Methode heißt, neuronales Netzwerk. Was ein neuronales Netzwerk genau ist und wie es aufgebaut ist, erkläre ich etwas später genauer. Jetzt nur so viel: ein neuronales Netzwerk ist der Versuch ein menschliches Gehirn am Computer nachzubauen.

Zum Modellieren eines neuronalen Netzwerks benutze ich das von vielen großen Firmen (zum Beispiel Google, eBay, AMD, Nvidia, Uber u.a.) genutzte Framework Tensorflow, mit dessen Anbindung der ebenso oft genutzten API (Programmierschnittstelle) Keras. Daraus folgt meine Fragestellung für meine Facharbeit: Kann ein Computer lesen lernen? - Auslesen eines Nummernschildes über ein neuronales Netzwerk unter Benutzung von Tensorflow und Keras.

Ich werde in dieser Arbeit der Schreibarbeit wegen und, weil dieser Begriff so in der Literatur benutzt wird, ein neuronales Netzwerk ein NN nennen.

Zur Darstellung eines NNs werde ich in dieser Arbeit folgendes Format verwenden: Größe des Input Layers/Größe des ersten Hidden Layer[/Größe des zweiten Hidden Layers/...]/Größe des Output Layers. Die Größe eines Layers beschreibt, wie viele Neurons in diesem sind[[1]](#footnote-1).

# Was ist ein NN?

Ein NN ist eine Ansammlung von Algorithmen, die Eingabedaten auf eine bestimmte Art und Weise klassifizieren sollen, ähnlich, wie es das menschliche Gehirn macht. Neuronale Netzwerke sind deswegen dem menschlichen Gehirn, beziehungsweise dessen Arbeitsweise, nachempfunden. Dabei kann ein NN Daten nur als Zahlen, die (meistens) in Form eines Vektors eingegeben werden, gefüttert werden. Das bedeutet, dass Daten, wie zum Beispiel Bilder, erst in solche Zahlen umgewandelt werden müssen.

Damit ein NN die Eingaben auch korrekt klassifizieren kann, muss es erst mal trainiert werden, ähnlich, wie ein Mensch auch erst lernen muss, um eine Aufgabe bewältigen zu können. Auf die verschiedenen Möglichkeiten, die es zum Trainieren gibt und wie das Trainieren funktioniert, komme ich später zurück. Im Generellen heißt lernen nur, dass sich das NN "merkt" wie bestimmte Daten klassifiziert wurden und das dann nachahmt. Wie dieses "merken" funktioniert, erkläre ich, wenn ich die Funktionsweise der einzelnen Komponenten eines neuronalen Netzwerks erkläre.

# Allgemeine Information zu NN

Über die Zeit haben sich verschiedene Arten an NNs entwickelt. Der Einfachheit halber benutze ich hier als Beispiel zum Erklären des Aufbaus und der Funktion ein "normales" NNs (auch "multilayer perceptrons"). Diese Art an NN ist eine der zuerst entwickelten. Sie entstand in den 1960er.

## Aufbau eines NNs

Ein "normales" NN besteht aus mehreren Layern[[2]](#footnote-2). Jeder Layer besteht aus ein oder mehr Neurons2. Jedes Neuron eines Layers ist mit jedem Neuron des nächsten Layers über eine Connection2 verbunden. Jedes Neuron besitzt einen Bias2, der anzeigt ab welcher Aktivierung das Neuron aktiviert sein soll und jede Connection besitzt eine Weight2. Im folgenden erkläre ich die einzelnen Komponenten noch einmal genauer. In einem NN sind alle Komponenten fest und können nicht verändert werden, außer der Weights und Biases. Nur durch das Verändern dieser beiden Komponenten wird ein NN "schlau". Da diese Vielfalt der Einstellungsmöglichkeiten sehr groß ist, kann man dies nicht per Hand bearbeiten, sondern braucht eine Trainingsfunktion, die dies übernimmt. Wenn man, zum Beispiel, ein einfaches 784/16/16/10 NN benutzten gibt es insgesamt 13.002 verschiedene Weights und Biases, die jeweils alle Werte aus den reellen Zahlen annehmen können.

Ein NN wird graphisch oft so dargestellt:

****

Abbildung 1: Graphische Darstellung NN[[3]](#footnote-3)

Bei dieser Darstellungsart kann man gut die einzelnen Neurons, Connections und Layers erkennen. Hieran wird auch deutlich woher der Begriff "deep learning" kommt. Das "deep" meint nichts anderes, als dass jedes Neuron eines Layers mit jedem Neuron des vorherigen und nächsten Layers verbunden ist.

### Layer

Ein Layer ist nichts weiter, als eine Aufbewahrung für mehrere Neurons. In einem "normalen" NN sind mindestens drei, meist eher 4 Layers zu finden. Diese teilen sich auf in ein "Input Layer"2, ein "Output Layer"2 und ein oder mehr "Hidden Layers"2. Das "Hidden" bedeutet nur, dass diese Layer weder Input noch Output Layer sind.

### Neuron

Ein Neuron ist ein Aufbewahrungsort für eine Zahl, die (bei einem "normalen" NN) zwischen null und eins liegt. Bei allen Neuronen, außer denen in dem Input Layer, wird die sog. Aktivierung (die gespeicherte Zahl) durch die, die Ausgabe einer Aktivierungsfunktion definiert, die als Eingabe die gewichtete Summe aller Eingabe an das Neuron plus den vom Neuron bestimmten Bias bekommt. Die Eingaben an ein Neuron sind die Connections von dem vorherigen Layer. Das Input Layer bekommt als Aktivierung die Werte der zu überprüfenden Daten, zum Beispiel die Graustufenwerte eines Bildes. Dabei hat jedes Neuron den Wert eines Pixels. Die Gewichtung der Eingaben kommt durch die Weights an den Connections zustande.

### Connection

Eine Connection ist eine Verbindung eines Neurons eines Layers zu einem Neuron des nächsten Layers. Dabei wird die Aktivierung des ersten Neuron, unter Berücksichtigung des Weights der Connection an das zweite Neuron übertragen.

### Bias

Ein Bias ist ein Indikator dafür, wie schnell ein Neuron aktiviert sein soll. Jedes Neuron besitzt einen Bias. Der Bias ist eine der beiden Komponenten, die in einem NN verändert werden können und somit das "Lernen", beziehungsweise "Merken" des NNs ausmachen.

### Weight

Eine Weight ist die Gewichtung einer Aktivierung bei der Bestimmung der Aktivierung des nächsten Neurons. Sie ist die zweite Komponente, die in einem NN verändert werden kann.

### Aktivierungsfunktion

Eine Aktivierungsfunktion ist dafür da, die vorläufige Aktivierung eines Neurons (gewichtete Summe plus Bias) in die Spannweite der Zahlen zwischen null und eins zu drücken. Eine oft genutzte Aktivierungsfunktion ist die Sigmoidfunktion. Dies ist definiert als:

Formel 1: Definition Sigmoidfunktion

Diese Funktion drückt einen Wert zwischen null und eins aus. Je kleiner der Wert, desto näher ist er an der Null. Je größer, desto näher an der Eins. Dies kann man dem Verlauf der Sigmoidfunktion gut entnehmen:

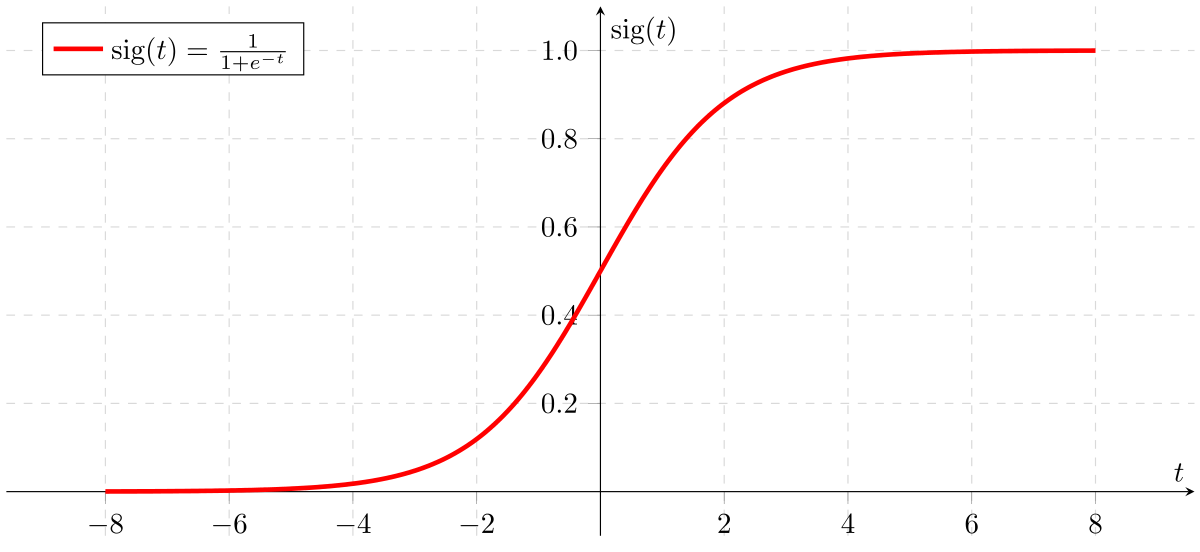


Abbildung 2: Sigmoidfunktion gezeichnet

### Konventionen zum Aufbau eines NNs

Um zu bestimmen wie viele Layers und Neurons ein NN braucht, kann man eine einfache Konvention zur Hilfe ziehen:

Tabelle 1: Regeln zur Bestimmung der Größe eines NNs

|  |  |
| --- | --- |
| Input Layer | Das Input Layer hat immer so viele Neurons, wie man braucht. Wenn man, zum Beispiel, Bilder klassifizieren will, braucht man so viele Neurons, wie Pixel in den Bildern sind, dass heißt n(Input) = Pixelbreite \* Pixelhöhe. |
| Output Layer | Im Output Layer braucht man so viele Neurons, wie Labels, den man den Daten zuschreiben will. Wenn man nur wissen will, ob auf einem Bild etwas Bestimmtes zu sehen ist, oder nicht, braucht man nur ein Neuron. Will man wissen welche Ziffer zu sehen ist, braucht man zehn. |
| Hidden Layer | Man benutzt normalerweise nur ein bis zwei Hidden Layer. In Ausnahmefällen auch drei. Viel mehr als das sollte man aber nicht benutzen. |

Zur Bestimmung, wie viele Neurons in den Hidden Layern man braucht gibt es drei Regeln:

Tabelle 2: Regeln zur Bestimmung der Anzahl der Neurons in den Hidden Layern

|  |  |
| --- | --- |
| 1. Regel | Die Anzahl der Neurons in den Hidden Layern sollte zwischen der Größe des Input Layers und des Output Layers. |
| 2. Regel | Die Gesamtanzahl an Hidden Neurons sollte kleiner sein, als die doppelte Größe des Input Layers. |
| 3. Regel | Die Gesamtanzahl an Hidden Neurons sollte ungefähr 2/3 der Gesamtgröße des Input und Output Layers. |

## Benutzung

Wie bereits erwähnt, kann ein NN Daten nur in Form von Zahlen annehmen. Diese Zahlen werden dann, so wie oben beschrieben, durch das NN gegeben. Am Ende gilt das Neuron im Output Layer mit der größten Aktivierung, als die Antwort des NNs. Damit das NN korrekte Antworten geben kann, muss es, wie schon gesagt, trainiert werden, dass heißt, es bekommt Beispieldaten, mit den richtigen Antworten und versucht dann die verschiedenen Einstellungsmöglichkeiten (Weights und Biases) so zu verändern, das die richtigen Antworten gegeben werden. Dieser Vorgang wird durch eine Trainingsfunktion verwirklicht.

### Trainingsfunktionen

Es gibt verschiedene Arten wie man ein NN trainieren kann. Die Meistbenutzten sind "supervised learning" und "unsupervised learning". Beim "unsupervised learning" bekommt das NN nur Daten und muss diese dann alleine klassifizieren. Beim "supervised learning" bekommt das NN Paare an Daten und Labeln (Klassifizierung der Daten) und versucht mit den Eingabedaten die zugehörige Klassifizierung nachzuahmen. Ich gehe im Folgenden auf das "supervised learning" ein, da ich diese auch in meinem späteren Beispiel benutze.

Das NN geht beim "supervised learning" die Trainingsdaten durch und bildet zu jedem seine Ausgabe. Diese wird dann mit dem sog. Label, also der vorgegebenen Klassifizierung insofern verglichen, als dass die Summe der quadrierten Differenz zwischen der Antwort des NNs und dem Label gebildet wird. Dies wird für jedes Beispiel im Trainingsdatenpacket gemacht. Aus diesen Werten wird dann das arithmetische Mittel gebildet. Der daraus entstehende Wert gibt an, wie schlecht das NN die Trainingsdaten klassifizieren kann. Dieser Wert wird durch eine sog. Cost Function2 bestimmt. Die hier beschriebene Cost Function heißt Mean Squarred Error2, oder kurz MSE.

Um aus dieser Cost Function lernen zu können, muss nun der Tiefpunkt der Cost Function ermittelt werden. Die Cost Function bezieht als Variablen die verschiedenen Weights und Biases. Um bei einer Funktion mit mehreren Variablen einen Tiefpunkt zu finden, kann man den negativen Gradienten[[4]](#footnote-4) nehmen. Ein Gradient, oder besser ein Gradientenvektor, beschreibt die Richtung in die man, von einem gewissen Punkt, gehen muss bei einer mehrdimensionalen Funktion, in die die Funktion am steilsten steigt. Dementsprechend fällt der Graph in die andere Richtung am meisten. Diese Methode heißt in der Literatur "Gradient descent".

Um diesen negativen Gradienten auszurechnen, benutzt man meistens einen sog. Optimizer2. Fast alle dieser Optimizer basieren auf dem Prinzip der Backpropagation2. Da die Backpropagation ein sehr umfangreiches Prinzip ist, erkläre ich dieses unter der nächsten Überschrift.

### Backpropagation

Zur Berechnung des Gradienten der Cost Function legt man einen Vektor an, in dem jedes Element dem Mittelwert der Ableitung nach einer Variable (Weight oder Bias) der Cost Function aller Trainingsdaten entspricht. Dabei ist man die Ableitung nach einer der Variablen eines Trainingsbeispiels wie folgt definiert:

Formel 2: Berechnung Ableitung Cost nacheinem Beispiel nach w^(L)

Dabei gilt:

Tabelle 3: Begriffserklärung der in Formel 2 genutzten Zeichen

|  |  |
| --- | --- |
|  | Cost Function nach einem Trainingsbeispiel |
|  | Weight der Connection zu dem betrachteten Neuron |
|  | Aktivierung des betrachteten Neuron |
|  |  |
|  | Aktivierung des Neurons an der anderen Seite der Connection mit |
|  | Bias des betrachteten Neuron |

Diese Berechnung muss für alle Trainingsdaten vollzogen werden und daraus wird der Mittelwert gebildet. Dieser entspricht dann den Gradienten der Cost Function an der Stelle von .

Die Backpropagation versucht erst die Aktivierungen der vorherigen Neurons zu bearbeiten. Erst wenn das nicht mehr geht, da der Algorithmus an dem Input Layer angekommen ist, werden diese Änderung an den Aktivierungen auf die Weights und Biases zurückgeführt.

Da dieser Vorgang sehr Zeitintensiv sein kann, teilt man meist das Trainingsdatenset in sog. Batches2 auf. Dann muss man pro Durchgang nur ein Batch durchgehen und nicht das gesamte Datenpacket. Desweiteren wird oft festgelegt, wie viele Epochen das NN trainiert werden soll, damit es nicht unendlich weiter läuft, da es kein Tiefpunkt der Cost Function findet.

### Anwendungsbeispiele

Neuronale Netze können in vielen verschiedenen Bereichen benutzt werden. Beispiele, in denen oft NNs benutzt werden, sind Tonerkennung (Sprach-zu-Text-Erkennung in beispielsweise Handys), Bilderkennung (erkennen, was auf einem Bild zu sehen ist), Bildauslesen (Bild-zu-Text-Erkennung, wie in dieser Arbeit) oder in der einfachsten Form, finden von Funktionsgleichungen zu bestimmten Wertepaaren.

# Was sind Tensorflow und Keras?

Im Folgenden gehe ich auf Tensorflow und Keras ein, da ich diese beiden Frameworks/APIs in dem Eigenanteil meiner Facharbeit benutze.

## Erklärung von Tensorflow und Keras

Tensorflow ist ein Framework zum Modellieren von NNs. Das bedeutet, dass andere Programmierer, in diesem Fall Programmierer von Google, hingesetzt haben und alles Mögliche, was jemand zum Modellieren eines NNs brauchen könnte, zusammen geschrieben haben und als Code zur Verfügung gestellt haben, damit andere Programmierer weniger Arbeit haben. Dabei haben diese Programmierer die API Keras, die es schon vorher gab und in etwa das gleiche ist, wie Tensorflow, ebenfalls eingebunden, damit alte Programme, die mit Keras gemacht wurden, in die Umgebung von Tensorflow eingebunden werden können, beziehungsweise andere Programmierer, die sich mit Keras schon auskennen, auch Tensorflow benutzen können, ohne viel Neues zu lernen. Dabei ist zu beachten, dass Keras ein eigenes Dateiformat eingeführt hat, um NNs zu speichern und zu einem anderen Zeitpunkt oder anderem Computer wieder zu laden und dort weiter zu benutzen, bzw. zu trainieren. Diese Möglichkeit besteht in Tensorflow natürlich, durch die Anbindung von Keras, auch. Man kann außerdem alte NNs genauso laden, wie man es vorher aus Keras gewöhnt war. Dabei bietet Tensorflow einige Vorteile gegenüber Keras, da es auf Keras aufbaut und damit erweitert.

## Wie wird mit Tensorflow ein NN modelliert?

In Tensorflow gibt es mehrere Möglichkeiten ein NN zu modellieren. Ich gehe im Folgenden nur auf die Methode mit Benutzung von Keras ein, da ich dies auch in meinem Eigenanteil benutzen werde. Außerdem benutze ich die Anbindung von Tensorflow in Python, da diese die Einzige ist, die vollständig ist.

Um ein sog. Model2, also eine Darstellung eines NNs zu erstellen benutzt man den Befehl tf.keras.models.Sequential(\*args). In den Klammern kann man direkt eine Liste an Layern angeben, muss dies aber nicht unbedingt tun. Um dann ein Layer hinzuzufügen benutzt man entweder eine Liste, wie gerade beschrieben, oder benutzt den Befehl model.add(arg). Diesem Befehl muss man in den Klammern ein Layer angeben. Es gibt in der Keras Layers API viele verschiede Möglichkeiten an Layers, die man hier einfügen kann. Die einfachste Möglichkeit wäre ein Dense Layer zu benutzen. Dense bedeutet, dass alle Neurons dieses Layers mit allen Neurons des nächsten Layers verbunden sind, genau so, wie wir es haben wollen. Eine komplette Zeile an Code um ein Layer hinzuzufügen sähe dann so aus: model.add(tf.keras.layer.Dense(16, input\_shape=(748,), activation=tf.nn.sigmoid). Die erste Zahl in den Klammern des Dense Befehls gibt an, wie viele Neurons in diesem Layer sein sollen, in diesem Beispiel sind es 16. der zweite Eintrag in der Klammer setzt fest wie viel Input Neurons dieses Layer haben soll. Dies muss nur beim ersten Layer angegeben werden und beschreibt damit das Input Layer. Der letzte Eintrag legt fest, dass wir als Aktivierungsfunktion die Sigmoidfunktion benutzen. Wie in der offiziellen Dokumentation von Tensorflow nachzulesen, kann man noch viel mehr Parameter in den Klammern übergeben, auf diese gehe ich jetzt aber nicht weiter ein.

Damit man das Model trainieren und damit arbeiten kann, muss man es erst einmal Kompilieren. dazu ruft man den Befehl model.compile(optimizer, loss) auf. In diesem Aufruf legt man fest, welcher Optimizer und welche Cost Function benutzt werden sollen. Welche Möglichkeiten Tensorflow dazu anbietet, ist wieder in der Dokumentation zu finden.

Um das Model zu trainieren, wird nur eine Zeile Code benötigt: model.fit(\*feature, \*label, \*dataset, \*epoch, \*steps\_per\_epoch). Dabei kann man entweder ein Tensorflow eigenes Dataset eingeben, oder eine Trainingsfeature- und Trainigslabelliste einzeln voneinander. Danach gibt man an, wie viele Epochen man trainieren will und wie groß die Batches sein sollen.

Jetzt kann man das NN entweder direkt benutzen, oder es abspeichern und später wieder laden.

# Vorbereitung der Daten

Bisher habe ich mich allgemein gehalten und den Hintergrund meiner Facharbeit erklärt. Im Folgenden werde ich mich konkret auf mein Eigenanteil beziehen. Dafür fange ich mit der Vorbereitung der Test- und Trainingsdaten an.

## Trainingsdaten-Set

## Testdaten

# Aufbau und Anbindung eines NNs zum Auslesen eines Nummernschildes

## Pythoncode

## Javacode

# Evaluation der Ausgabe des Programmes

# Fazit

# Literaturverzeichnis

## Bildquellen

Tensorflow Logo:

Quelle: <https://de.wikipedia.org/wiki/Datei:Tensorflow_logo.svg>

Lizenz: [Creative Commons](https://creativecommons.org/licenses/by-sa/4.0/deed.de)

Urheber: "FlorianCassayre"

Keras Logo:

Quelle: <https://de.wikipedia.org/wiki/File:Keras_Logo.jpg>

Lizenz: keine

Urheber: "Francois Chollet"

Tensorflow und Keras:

Quelle: <https://bit.ly/2BMazAF>

Lizenz: keine Angabe

Urheber: keine Angabe

Kamera:

Quelle: <https://www.flaticon.com>

Lizenz: [Creative Commons](https://creativecommons.org/licenses/by-sa/4.0/deed.de)

Urheber: [Freepik](https://www.freepik.com)

Kopf:

Quelle: <https://www.flaticon.com>

Lizenz: [Creative Commons](https://creativecommons.org/licenses/by-sa/4.0/deed.de)

Urheber: [Smaschicons](https://www.flaticon.com/authors/smashicons)

## Textquellen

*A Beginner's Guide to Neural Networks and Deep Learning*. 2018. 28. Februar 2019. <https://skymind.ai/wiki/neural-network>.

*How to choose the number of hidden layers and nodes in a feedforward neural network*. 2010-2015. 29. Februar 2019. <https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw>.

Nielson, Michael. *Neural Networks and Deep Learning*. 2015. 28. Februar 2019. <http://neuralnetworksanddeeplearning.com>.

# Abbildungsverzeichnis

Abbildung 1: Graphische Darstellung NN 4

Abbildung 2: Sigmoidfunktion gezeichnet 6

# Tabellenverzeichnis

Tabelle 1: Regeln zur Bestimmung der Größe eines NNs 6

Tabelle 2: Regeln zur Bestimmung der Anzahl der Neurons in den Hidden Layern 6

Tabelle 3: Begriffserklärung der in Formel 2 genutzten Zeichen 8

# Formelverzeichnis

Formel 1: Definition Sigmoidfunktion 5

Formel 2: Berechnung Ableitung Cost nacheinem Beispiel nach w^(L) 8

# Anhang

**Eigenständigkeitserklärung**:

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die im Literaturverzeichnis angegebenen Hilfsmittel verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken und aus den genutzten Internetquellen als solche kenntlich gemacht habe. Mir ist bekannt, dass die Facharbeit einer externen Plagiatskontrolle unterzogen wird.

Sofern sich – auch zu einem späteren Zeitpunkt – herausstellen sollte, dass die Arbeit oder Teile davon nicht selbstständig verfasst wurden, so wird die Arbeit auch nachträglich mit null Punkten bewertet.

Ort, Datum Unterschrift

1. Erklärung siehe 3.1.1 [↑](#footnote-ref-1)
2. Ich benutze den englischen Begriff, da dieser auch in der Literatur so genutzt wird [↑](#footnote-ref-2)
3. entnommen aus Neural Networks and Deep Learning [↑](#footnote-ref-3)
4. Zum Verständnis: Ein Gradient einer Funktion ist ein Vektor bestehend aus allen Ableitungen der Funktion nach den verschiedenen Variablen der Funktion. [↑](#footnote-ref-4)