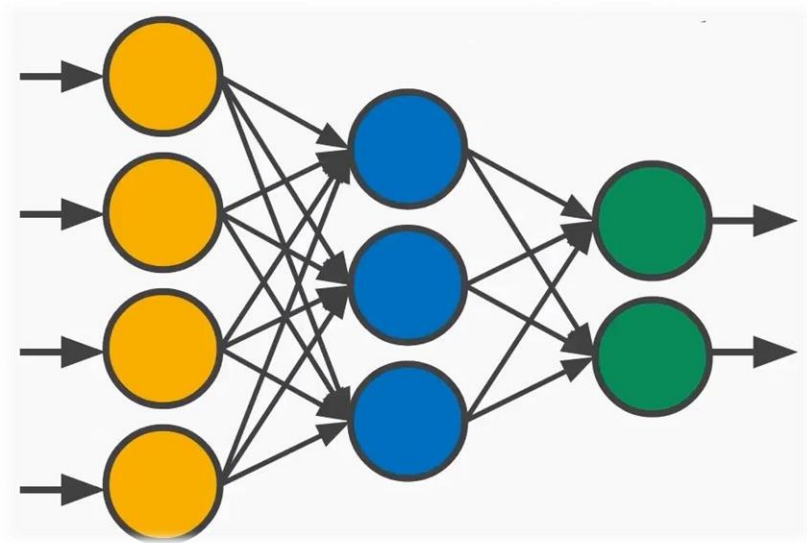


Künstliche Neuronale Netzwerke



Reuter Max

Eifes Béatrice

4C1

Lycée Ermesinde

2018-2019

Inhaltsverzeichnis

EINLEITUNG.....	2
EINFÜHRUNG	3
Künstliche Intelligenz, Maschinelles Lernen und Deep Learning	3
Idee hinter dem KNN.....	3
Anwendungsbereiche eines künstlichen neuronalen Netzwerkes	4
THEORETISCHE BASIS	5
Künstlichen Neuronen.....	5
Funktionsweise des neuronalen Netzes	5
Aufbau	6
Verbindungen zwischen Knoten	6
Training- und Testdaten	6
Underfitting, Sweet Spot und Overfitting	7
Sigmoidfunktion.....	7
Matrizendarstellung	9
PROGRAMMIERUNG EINES KÜNSTLICHEN NEURONALEN NETZWERKES	10
Aufsetzen des künstlichen neuronalen Netzwerkes	10
Initialisierung	10
Training.....	11
Abfrage	12
Erstellung die künstlichen neuronalen Netzwerke.....	12
Training des künstlichen neuronalen Netzwerkes	13
Testen des künstlichen neuronalen Netzwerkes	15
SCHLUSSWORT	16
QUELLEN	17
Bilderquellen:.....	17

Einleitung

Seit Jahrhunderten ist die Menschheit auf der Suche nach neuen Technologien und seit Jahrzehnten wird nach einer intelligenten Maschine geforscht. Die erste Präsenz einer solchen Intelligenz fand am 15. März 2016 seinen Platz in der Geschichte. An diesem Tag hat die künstliche Intelligenz Alpha Go den Weltmeister im hochkomplexen Spiel Go, in 4 von 5 Fällen geschlagen. Alpha Go wurde von DeepMind, ein zu Google gehöriges Unternehmen, entwickelt. Nach diesem Ereignis wurde Alpha Go von seinem Nachfolger Alpha Go Zero geschlagen. Seitdem sind die Themen künstliche Intelligenz und künstliche neuronale Netzwerke sehr präsent. Heutzutage sind die Technologien gar nicht mehr wegzudenken, die Meldungen über Anwendungen in verschiedensten Bereichen überschlagen sich. Egal ob es sich bei der Post um die Erkennung der Adressen auf Briefen handelt, bei Google bei der Bilderkennung Gebrauch findet oder ob es sich bei Nvidia um die automatische Färbung von schwarz-weiß Bildern dreht, die Entwicklung der künstlichen neuronalen Netzwerke ist im vollen Gange und das Potenzial dieser überaus machtvollen Technologie ist noch lange nicht ausgeschöpft.

Doch was ist ein künstliches neuronales Netzwerk? Wenn es um das Lösen von einfachen Gleichungen geht, kann niemand einem Computer etwas vormachen. Einfache Rechnungen werden in Bruchteilen von Sekunden gelöst. Dies bietet dem PC viele Anwendungsbereiche, das Streamen von Videoinhalten ist auch nichts anderes als das Dekodieren von Einsen und Nullen. So gesagt ist ein Supercomputer nichts anderes als ein hochgezüchteter Taschenrechner. Im Punkt „Bilderkennung“ hinkt der Computer uns aber hinterher. Wir müssen uns nicht anstrengen, um eine Katze von einem Glas zu unterscheiden. Für einen Rechner ist dies aber eine hoch komplexe Aufgabe, welche über herkömmliche Gleichungen gar nicht möglich ist. Deshalb wurden künstliche neuronale Netzwerke erfunden, mit deren Hilfe man dem Computer beibringen kann, wie man verschiedenen Objekte erkennt.

In diesem „Travail personnel“ werde ich ein eigenes künstliches neuronales Netzwerk programmieren und diesem beibringen, wie man handgeschriebene Zahlen erkennt.

Einführung

Künstliche Intelligenz, Maschinelles Lernen und Deep Learning

Künstliche Intelligenzen, maschinelles Lernen, Deep Learning und künstliche neuronale Netzwerke sind Begriffe, die oft verwechselt, missverstanden und falsch gebraucht werden. Um ein wenig Klarheit zu schaffen, werde ich die Wörter einmal genauer erläutern.

Künstliche Intelligenz:

„Eine künstliche Intelligenz ist ein Programm, das fühlt, denkt, handelt und sich anpassen kann.“ (Bayer T.)

Maschinelles Lernen:

„Das maschinelle Lernen ist ein Algorithmus, der über die Zeit besser wird, je mehr Daten er analysiert.“ (Bayer T.)

Deep Learning:

„Das Deep Learning ist eine Teilmenge des maschinellen Lernens, die dank neuronaler Netze unglaubliche Datenmengen verarbeitet und aus ihnen lernt.“ (Bayer T.)

Künstliche Neuronale Netzwerke:

„Künstliche Neuronale Netze (KNN) sind selbständig lernende Computerprogramme. Sie können schnell sehr komplizierte Regelmäßigkeiten und Zusammenhänge in großen Datenmengen erkennen.“ (OnPulson)

Wie man sieht, bauen alle Begriffe aufeinander auf, sind aber auf gar keinen Fall dasselbe.

Idee hinter dem KNN

Ein Computer ist in seiner Basis immer nur eine hochleistungsfähige Rechenmaschine. Arithmetische Aufgaben absolviert ein Rechner in Bruchteilen von Sekunden. In diesem Gebiet sind sie so weit entwickelt, dass Menschen für solche Rechnungen komplett irrelevant geworden sind. In den Bereich der Arithmetik fallen auch Aufgaben wie Steuern berechnen, Diagramme zeichnen usw. Ebenso ist das Streamen von digitalen Inhalten nichts anderes als das Umrechnen, auch genannt Decodieren, von einer enorm großen Anzahl von Nullen und Einsen. Kurzgefasst, ein Computer benutzt noch immer Operatoren, die nicht komplexer sind als die, welche wir in unser Grundschulzeit gelernt haben.

Schnell addieren benötigt jedoch keine Intelligenz. Einem Menschen scheint es zwar schwer, Zahlen schnell fehlerlos hintereinander zu multiplizieren, jedoch ist es für einen PC nur das Folgen von einfachen Anweisungen.

Die Idee hinter den künstlichen neuronalen Netzen ist, die Aufgaben, die einem Menschen unfassbar einfach fallen, ebenso als Rechner lösen zu können. Nehme man ein einfaches Portrait von einem Menschen: für eine Person ist es selbstverständlich, dass es sich bei dem abgebildeten Motiv um einen Menschen handelt. Für einen Computer ist dies nicht

selbstverständlich. Dieser muss tausende, Millionen oder gar Milliarden Bilder analysieren, um einen Menschen definieren und erkennen zu können.

Wenn man zum Beispiel eine Frage stellen würde, was den Menschen optisch von einem Tier unterscheidet, würde ein willkürlich Befragter verschiedene einzigartige Merkmale aufzählen und den Unterschied klar erkennen. Der Rechner jedoch versucht diese Frage in Form von wiederkehrenden Mustern zu lösen, in dem er tausende Bilder von Menschen, sowie auch Tieren analysiert.

Dies fordert eine Art von Intelligenz vom Computer. Ohne die Funktionsweise von künstlichen neuronalen Netzen könnten nicht einmal die leistungstärksten Rechner der Welt eine handgeschriebene Null von einer Eins unterscheiden.

Da Computer initial auf Elektronik basieren, gilt es neue Algorithmen zu finden, die solche schwierige Probleme lösen können. Selbst wenn dies nicht immer zu 100% gelingen kann, dann immer noch gut genug, um den Anschein von einer Intelligenz zu erzeugen. Ein Computer muss dann wie ein Mensch durch Erfahrungen, im Falle des Computers durch Analyse, lernen um somit verlässlicher zu werden und im optimalen Fall sogar fehlerfrei zu werden.

Anwendungsbereiche eines künstlichen neuronalen Netzwerkes

Ein künstliches neuronales Netzwerk ist nur teils theoretisch, die Praxis bleibt immer das Ziel eines solchen Netzes, deshalb wollte ich veranschaulichen, was man alles mit einem KNN anrichten kann. Ein Großteil der bestehenden Unternehmen nutzt solche Netze für verschiedenste Bereiche. Die Post nutzt ein solches um Adressen auf Briefen zu identifizieren, Google nutzt diese für jegliche Form der Datenerkennung, das heißt für Bild-, Muster-, Spracherkennung, Qualitätssicherung und Automatisierungstechnik. Ein genaueres Beispiel ist das Institut für Maschinenwesen der TU Clausthal. Diese Forscher haben im Rahmen des Sonderforschungsbereichs „Fertigen in Feinblech“ ein künstliches neuronales Netz programmiert. Dieses half ihnen das Rückfederungsverhalten verschiedener Blechteile vorherzusagen. Ebenso kann man diese Netze im Bereich Regression und Klassifikation nutzen.

Theoretische Basis

Künstlichen Neuronen

Der Hauptstruktur des KNNs bilden die Knoten, auch noch als Neuronen, Einheiten oder Units bekannt.

Es gibt 3 Haupt-Arten von Units:

Die **Input-Units**: Diese Neuronen haben die Aufgabe die Eingangssignale aufzunehmen und sie fast unverarbeitet an andere Knoten weiterzugeben.

Die **Hidden-Units**: Diese Neuronen befinden sich zwischen den Input und den Output Schichten. Diese Neuronen sind maßgeblich an der Verarbeitung der Daten beteiligt.

Die **Output-Units**: Die Output-Units geben die Signale an die Außenwelt ab.

Um die Funktionsweise eines künstlichen neuronalen Netzes zu verstehen, ist das Verstehen der Funktionsweise der künstlichen Neuronen essentiell.

Ein künstliches Neuron ist nichts anderes als eine mathematische Funktion. Diese nimmt x-beliebig viele Zahlen von der vorherigen Schicht als Eingabe, verarbeitet diese und gibt eine Zahl als Ausgabe. Die Eingaben können nicht direkt angenommen werden, sie werden noch gewichtet. Dazu kommt noch ein Bias, welches eine lineare Funktion darstellt, die zu den Wichtungen hinzugefügt wird.

Das Ergebnis, das man durch die Verarbeitung der Inputs herausbekommt, wird in eine Aktivierungsformel gesteckt. In meinem künstlichen neuronalen Netz benutze ich die Sigmoidfunktion, auf diese werde ich im späteren Verlauf der Arbeit genauer eingehen.

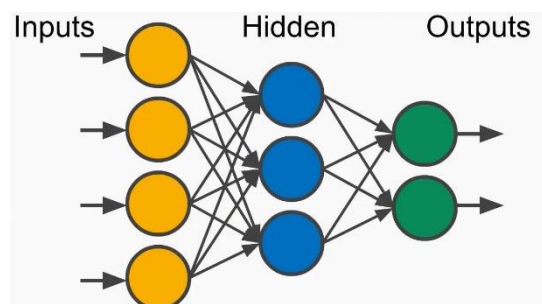
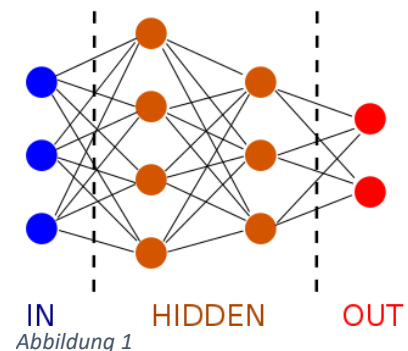
Nach diesen Schritten bekommt man eine verarbeitete reelle Zahl heraus, die man dann in die nächste Schicht von Neuronen einspeisen kann.

Funktionsweise des neuronalen Netzes

Die vorhin beschriebenen Neurone werden jetzt in einer Vielzahl in ein Netz eingesetzt. In meinem Fall setze ich die Neuronen in ein Feed-Forward Netz, das heißt die Ausgabesignale werden nur nach vorne weitergegeben.

Damit ein Programm lernen kann, muss natürlich irgendwo sein jetziger Wissensstand gespeichert sein. In einem KNN liegen diese Speicherungen in

den Kanten vor, da deren Gewichtung nach jedem Trainingsdurchlauf optimiert werden. Die Kanten sind die Verbindungen zwischen den Neuronen, in diesem Fall, die Striche zwischen den Punkten.



Wenn man einem KNN Eingaben vorgibt und ihm die korrekten Ausgaben vorgibt, versucht es die korrekte Ausgabe zu erzielen. Passiert dies jedoch nicht, schaut das Netz, wo der Fehler lag, analysiert wie die Signale hätten verlaufen müssen, um das richtige Ergebnis zu erzielen. So werden die Gewichte nach jedem Durchlauf angepasst.

Ein weiterer Vorteil der KNNs ist, dass man nicht genau wissen muss, was in der versteckten Schicht passiert. Man sagt dem Netz, was es als Eingabe bekommt und was es als Ergebnis erhalten soll. Wie es das macht, entscheidet das Netz selbst, so dass, wenn man kein Ergebnis mehr vorgibt, das Netz die Eingabe erhält und den eigenen Weg kennt.

Aufbau

Der Aufbau eines KNNs ist eigentlich sehr simpel. Ein Netz besteht meistens aus einer Eingabeschicht, einer oder mehreren versteckten Schichten und einer Ausgabeschicht. Angemerkt sei, dass es Variationen im Aufbau gibt, je nachdem welche Art von künstlichen neuronalen Netzen verwendet wird. Diese verschiedenen Schichten bestehen aus Neuronen, die untereinander und zwischen den Schichten durch Kanten verbunden werden.

Ein neuronales Netz muss jedoch nicht immer nur aus drei Schichten bestehen, die nacheinander ausgelöst werden. Die KNNs können viele verschiedene Strukturen annehmen. Die bekanntesten Strukturen sind jedoch die Feedforward-Netze und die Feedback-Netze. Das Feedforward-Netz ist das vorhin beschriebene klassische Netz, welches die Schichten nacheinander auslöst. Das Feedback-Netz sieht vom Aufbau her auch noch klassisch aus, kann jedoch während der Ausführung auf eine bereits ausgelöste Schicht zurückgreifen. Das heißt, bildlich erklärt, kann das Signal noch einmal zurückgehen und sich danach wieder nach vorne bewegen. Abgesehen von den zwei Strukturen, gibt es noch viele weitere Strukturen, welche auch auf verschiedenen Funktionsweisen basieren.

Verbindungen zwischen Knoten

Die Verbindungen zwischen den einzelnen Units beschreiben, welchen Einfluss ein Neuron auf das andere hat. Die Stärke der Verbindung wird durch Gewichte ausgedrückt. Je größer das Gewicht, je größer der Einfluss.

Man unterscheidet:

- Das positive Gewicht
- Das negative Gewicht
- Das Gewicht von Null

Training- und Testdaten

Ein fertiges Programm kann man nicht immer direkt reale Aufgaben bewältigen lassen und auch gar nicht mit der Realität vergleichen. Das angewendete Modell verstehen wir Menschen oftmals nicht. Das einzige, was wir Menschen kennen, sind die Eingaben und die richtigen Ausgaben. Dies ist äußerst praktisch, da wir das Modell nicht verstehen müssen und wir somit Programme entwickeln können, die unsere Kompetenzen weit übertreffen.

Um unser künstliches neuronales Netz zu trainieren, nutzen wir Trainingsdaten, um ein Modell zu erschaffen. Haben wir unser Netz dann genug trainiert, werden neue, bisher unbekannte

Daten eingespeist: die Testdaten. Mit dieser Methode kann man die Genauigkeit des Netzes feststellen.

Underfitting, Sweet Spot und Overfitting

Wenn unser Programm während des Trainingszeitraums nicht richtig gelernt hat und keine guten Ergebnisse im Training, wie auch im Test, aufbringt, hat man es mit *Underfitting* zu tun. Im Groben heißt das, dass das Programm einfach nicht genug an die Daten angepasst wurde. Das Resultat des *Underfittings* ist *High Bias*, das Ergebnis ist von Grund auf schlecht.

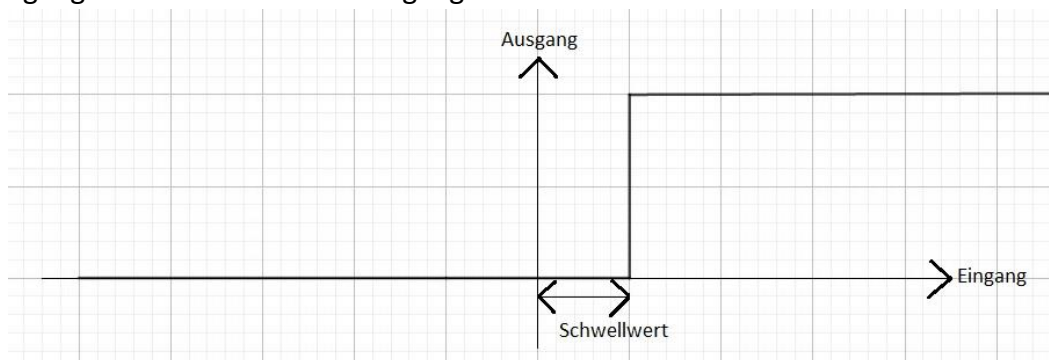
Beim *Sweet Spot* wurde das Programm besser an die Trainingsdaten angepasst, somit geht der Fehler bei den Trainings-, sowie auch Testdaten zurück. Da Test-, und Trainingsdaten ungefähr gleich gut sind, hat man hier den *Sweet Spot* erreicht.

Wenn man das Programm noch weiter auf die Trainingsdaten anpasst, dann wird der Fehler beim Training zwar geringer, doch der Fehler der Testdaten geht rauf, da das Programm zu stark auf das Training zugeschnitten wurde.

Sigmoidfunktion

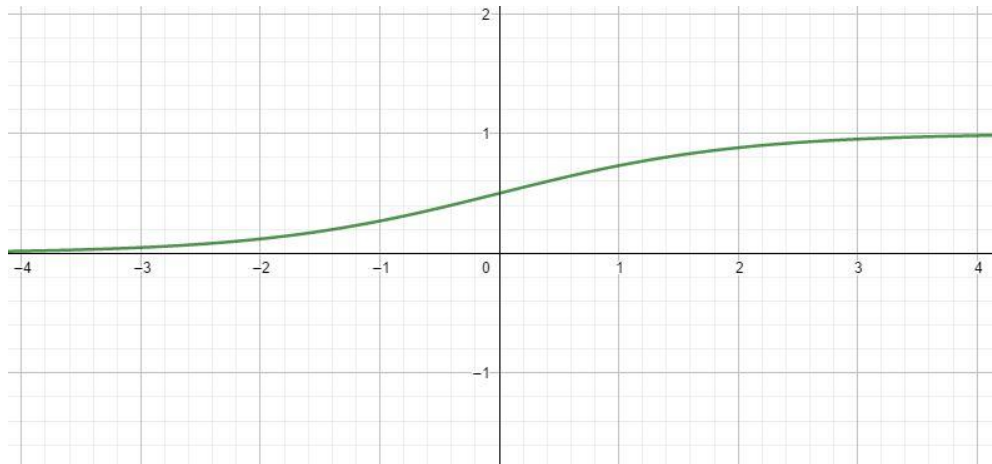
Es wurde bewiesen, dass Neuronen nicht bei jedem Signal reagieren. Erst wenn das Signal groß genug ist, feuern die Neuronen. Dies kann man sich wie einen Schwellenwert vorstellen und dient dazu, dass kleine Rauschsignale nicht durchdringen. Ziel ist, dass die großen gewollten Signale durchkommen.

Um dieses Konzept auf das KNN zu übertragen, erschafft man eine Aktivierungsfunktion. Eine Aktivierungsfunktion ist eine Funktion, die ein Eingangssignal nimmt und daraus ein Ausgangssignal bildet. In diesem Vorgang wird aber eine Art Schwellenwert berücksichtigt.



Es gibt eine Menge solcher Funktionen, die wohl einfachste ist die Stufenfunktion. Die Abbildung zeigt wie bei kleineren Eingabewerten die Ausgabe 0 ist, doch sobald der Schwellenwert erreicht ist, geht die Ausgabe sprunghaft nach oben.

Diese sprunghafte Aktivierung ist nicht natürlich, deshalb wird bei meinem KNN die Sigmoidfunktion genutzt. Diese Funktion bildet einen s-förmigen Anstieg. Dies macht das Ganze realistischer.



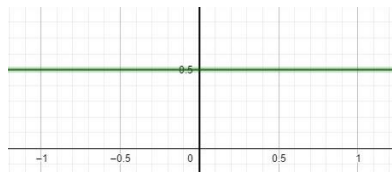
Die Sigmoidfunktion, oder logistische Funktion ist folgendermaßen definiert:

$$y = \frac{1}{1 + e^{-x}}$$

Um diese Funktion zu veranschaulichen zeige ich ein Beispiel. X ist gleich 0, deshalb wird der Ausdruck e^{-x} gleich 1. Das heißt $y = \frac{1}{1+1}$ und dies wird $\frac{1}{2}$, also schneidet die Sigmoidfunktion die y-Achse bei 0,5.

$$f : y = \frac{1}{1 + 2.72^0}$$

$$\rightarrow y = 0.5$$



Diese Formel ist eigentlich sehr simpel: die Eingabe x wird negiert und e zur Potenz dieses -x erhoben. In diesem Fall ist e die Eulersche Zahl, ihr numerischer Wert beträgt $e = 2,71828...$ Diese Zahl ist transzendent und somit ist sie auch eine irrationale reelle Zahl. Durch das Dividieren der 1 durch $1 + e^{-x}$ wird der Kehrwert des Gesamtausdrucks erstellt. Wird x durch eine Zahl ersetzt, bekommt man den Ausgabewert y.

In der späteren Anwendung wird mehr als nur eine Eingabe pro Neuron verwertet. Die Eingaben werden untereinander addiert. Das heißt z.B. $x = a + b + c$ und damit wird die Sigmoidfunktion berechnet $y(x)$.

Matrizendarstellung

Die Matrizen sind eines der wichtigsten Elemente, wenn es um künstliche neuronale Netze geht. Diese haben nämlich den Vorteil, dass sie ein KNN mathematisch darstellen kann. Die Matrix W enthält Werte w_{xy} : x steht für die Zeile und y für die Spalte. Jeder Wert w_{xy} steht für ein Gewicht. Durch die Matrizenrechnung ist es sehr einfach, die Gewichte zu aktualisieren und in der Matrix zu speichern.

Für jede Schicht ab 2 braucht man eine Matrix mehr, um das ganze KNN darzustellen, das heißt im Falle eines KNN ohne versteckte Schicht, könnte man das ganze Netzwerk innerhalb von einer Matrix beschreiben.

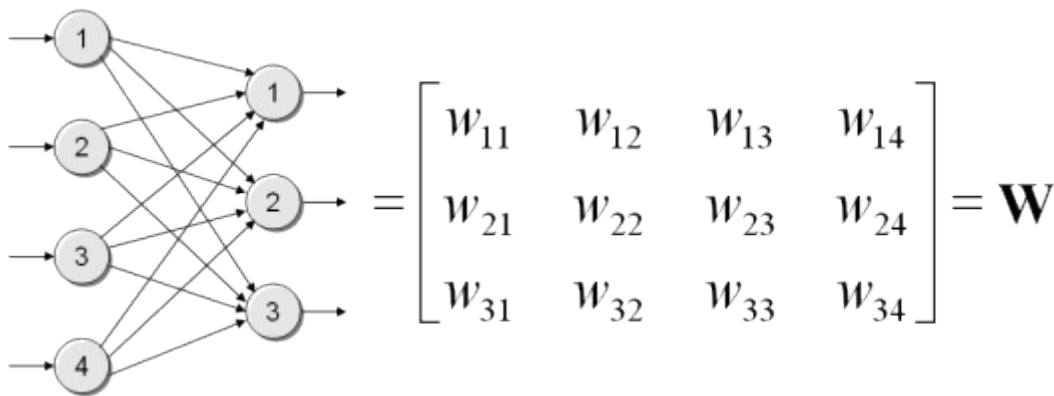


Abbildung 3 <http://www.neuronalesnetz.de/nnbilder/neuronennetze/matrix.gif>

Programmierung eines künstlichen neuronalen Netzwerkes

In diesem Kapitel dokumentiere ich den Weg zu meinem eigenen KNN. Dieses soll handgeschriebene Zahlen erkennen und dem Computer verständlich machen, so dass man digitale Zahlen als Ausgabe bekommt.

Aufsetzen des künstlichen neuronalen Netzwerkes

Der erste Schritt zum funktionierenden künstlichen neuronalen Netzwerk ist natürlich das Aufsetzen der Basis.

Das Netzwerk setzt sich aus drei Teilen zusammen: Initialisierung, Training und Abfrage. Die Initialisierung ist dafür zuständig Gewichte zu verteilen und fundamentale Variablen zu definieren, das Training soll die Gewichte stärken und die Abfrage soll schlussendlich handgeschriebene Zahlen in digitale Zahlen umwandeln können.

Aus diesen drei Teilen erschaffe ich die Grundstruktur. Diese bietet nun die Basis für den restlichen Code des KNN.

```
1  #Definition der Neuronalen Netzwerk Klasse
2  class neuralNetwork:
3
4      #Initialisierung
5      def __init__
6      pass
7
8      #Training
9      def train
10     pass
11
12     #Training
13     def train
14     pass
```

Initialisierung

Die Funktion „Initialisierung“ trägt den Namen `__init__` und nimmt 4 Argumente. Diese wären „self“, „input nodes“, „hidden nodes“, „output nodes“ und „learning rate“. Diese Parameter bestimmen das Ergebnis und sind nach Belieben veränderbar.

In dieser Funktion definiere ich die Anzahl der Knotenpunkte in jeder Schicht. Auf das genaue Quantum der Neuronen werde ich im späteren Verlauf der Dokumentation eingehen.

Als Nächstes werden die Matrizen mit den Verbindungs-Gewichten bestimmt. Um dies zu vollenden, nutze ich die Bibliothek Numpy. Der Vorteil an Bibliotheken ist, dass man oftmals sehr große Codes in wenigen Worten implementieren kann. Diese bieten mir den nötigen Code um eine zufällige Zahl zwischen 0 und 1 zu generieren. Nun wird, abhängig von der Knotenpunkt-Anzahl, eine Matrix erstellt und anschließend ausgerechnet. Da man berücksichtigen muss, dass die Gewichte nicht nur über positive Werte disponieren können, subtrahiere ich 0.5 von den willkürlichen Zahlen um diesen Wert von 0 bis 1 zu -0,5 und 0,5 umzuwandeln.

Als nächste determiniere ich die Lernrate. Diese kontrolliert die Anpassung der Gewichte in Bezug auf die Verlustgradienten, das heißt, sie legt die Größe der Schritte fest, mit welchen wir uns an das optimale Gewicht herantasten.

Die Initialisierung wird durch die Sigmoid-Aktivierungsfunktion beendet. Ich werde darauf nicht mehr genauer eingehen, da ich dies in einem früheren Kapitel bereits behandelt habe. Für diese Aufgabe gibt es ebenfalls ein Code, der in einer Bibliothek inbegriffen ist. Diesen Code findet man in der Funktion Lambda. Lambda wird genutzt, falls man eine Funktion erstellen möchte, die man später nicht mehr braucht und man ihr deswegen keinen Namen geben möchte. Ein weiterer Vorteil ist, dass man die Funktion sofort in einer Variablen integrieren kann.

Im unten gezeigten Code findet man immer wieder „self.“, dies dient nur dazu, dass jedes Objekt in der zuvor definierten Klasse „neuralNetwork“ diese Variable als Standard haben wird. Dies ist nötig, damit man in späteren Funktionen auf diese Variablen zugreifen kann.

```
def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate):
    # Definition der Anzahl der Knotenpunkte in jeder input, hidden und output Schicht
    self.inodes = inputnodes
    self.hnodes = hiddennodes
    self.onodes = outputnodes

    # Verbindungs-Gewichtungsmatrizen, wih und who
    # Gewichte in dem Feld sind w_i_j, unterdessem ist der Link vom Knotenpunkt i zum Knotenpunkt j in der nächsten Schicht
    # w11 w21
    # w21 w22 usw.
    self.wih = (numpy.random.rand(self.hnodes, self.inodes) - 0.5 )
    self.who = (numpy.random.rand(self.onodes, self.hnodes) - 0.5 )

    #Lernrate
    self.lr = learningrate

    # Aktivierungs Funktion = Sigmoid Funktion
    self.activation_function = lambda x: scipy.special.expit(x)

pass
```

Training

Im vorherigen Kapitel wurde die Initialisierung fertiggestellt, das heißt, dass die Knotenpunkt-Anzahl definiert und die Gewichte erstellt wurden. Im Training gilt es die Verbindungen zu optimieren, da die Werte anfangs randomisiert wurden.

Als kleine Randnotiz möchte ich hinzufügen, dass ich die Abfrage vor dem Training erstellt habe, doch es ist logischer, es in dieser Reihenfolge zu dokumentieren.

Das Training ist in 2 Teile unterteilt. Erstens, die Erzeugung einer Ausgabe. Dieser Schritt unterscheidet sich recht wenig von der, später kommenden, Abfrage. Zweitens, die Gegenüberstellung der Ausgabe und der gewünschten Ausgabe und die daraus resultierende Anpassung der Gewichte.

Bevor man das Training mit den Datensätzen starten kann, muss man die Dateien bearbeiten. Auf das Format der Datensätze werde ich im späteren Verlauf der Dokumentation noch einmal eingehen. Die einzige wichtige Information im Moment ist, dass das Bild, mit der handgeschriebenen Zahl, durch eine Liste aus vielen Zahlen repräsentiert wird. Diese Liste wird in ein 2D Feld umgewandelt, um es richtig verarbeiten zu können. Um dies umsetzen zu können, greife ich wieder auf die Numpy Bibliothek zurück. Aus dieser nehme ich mir die Funktionen „array“ und „ndmin“.

„array“ ist dafür zuständig, ein Feld zu kreieren und „ndmin“ gibt die Mindestanzahl der Dimensionen des Feldes an.

Im nächsten Schritt werden die Signale der versteckten Schicht berechnet. Dies passiert durch „numpy.dot“. Diese Funktion führt eine Matrizenrechnung aus, als Inputs nimmt sie die im vorherigen Kapitel definierte self.wih und die vorhin kreierte Input Variable. Durch die Hilfe der Selbstaktivierungsformel werden die Outputs berechnet. Dasselbe wird nochmal für die Ausgabeschicht wiederholt.

Zwecks Trainings wird die Ausgabe mit dem gewünschten Wert abgeglichen und in der nächsten Zeile wird der Fehler der versteckten Schicht „berechnet“. Der „hidden_error“ wird durch eine Matrizenrechnung berechnet. Die letzten 2 Zeilen beschreiben die Aktualisierung der Gewichte zwischen den Neuronen. Das Programm verbessert sich in kleinen Schritten, um sich nicht aus Versehen zu verschlechtern.

```
#Training
def train(self, inputs_list, targets_list):
    #Umwandlung der Input Liste in ein 2D Feld
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    #Berechnet Signale in die versteckte Schicht
    hidden_inputs = numpy.dot(self.wih, inputs)
    #Berechnet die Signale, welche aus der versteckten Schicht entstehen
    hidden_outputs = self.activation_function(hidden_inputs)

    #Berechnet Signale in die output Schicht
    final_inputs = numpy.dot(self.who, hidden_outputs)
    #Berechnet die Signale, welche aus der output Schicht entstehen
    final_outputs = self.activation_function(final_inputs)

    #Der Fehler ist das Ziel minus der gewünschte Wert
    output_errors = targets - final_outputs

    #Der Fehler der versteckten Schicht ist der output_error, geteilt durch die Gewichte, rekombiniert in der versteckten Schicht
    hidden_errors = numpy.dot(self.who.T, output_errors)

    #Aktualisierung der Gewichte der Verbindungen zwischen der versteckten und der output Schicht
    self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))
    #Aktualisierung der Gewichte der Links zwischen der versteckten und der input Schicht
    self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))
    pass
```

Abfrage

Die läuft nach dem exakt gleichen Schema wie das Training ab. Die einzigen Punkte, in denen sie sich unterscheiden, sind die Aktualisierung und die Umrechnung der Zielwerte. Diese beiden Punkte werden nicht mehr gebraucht, da man dem Netzwerk nun vertrauen muss und man es dadurch nicht mehr verbessern kann. Dies heißt aber nicht, dass man keine Kontrolle mehr über das neuronale Netz hat! In einem späteren Kapitel wir die Verifikation der Resultate geschildert.

```
#Abfrage
def query(self, inputs_list):
    # Umwandlung der Inputs in ein 2d Feld
    inputs = numpy.array(inputs_list, ndmin=2).T

    #Berechnet Signale in die versteckte Schicht
    hidden_inputs = numpy.dot(self.wih, inputs)
    #Berechnet die Signale, welche aus der versteckten Schicht entstehen
    hidden_outputs = self.activation_function(hidden_inputs)

    #Berechnet Signale in die output Schicht
    final_inputs = numpy.dot(self.who, hidden_outputs)
    #Berechnet die Signale, welche aus der output Schicht entstehen
    final_outputs = self.activation_function(final_inputs)

    return final_outputs
```

Erstellung die künstlichen neuronalen Netzwerke

Bevor das Training des künstlichen neuronalen Netzes beginnen kann, müssen noch verschiedene Variablen definiert werden.

Zuerst wird der Eingabenschicht eine Anzahl von 784 Neuronen zur Verfügung gestellt. 784 - da die Zahlen in einem Feld von 28 x 28 eingetragen wurden. Die Anzahl der Neuronen in der versteckten Schicht beträgt 200, diese Zahl kann man nach Belieben variieren. Dies kann sich negativ, sowie positiv herausstellen. Die Ausgabeschicht enthält 10 Neuronen, da das Ergebnis eine Zahl zwischen 0 und 9 sein muss.

Als nächstes wird die Lernrate definiert. Diese beschreibt in wie großen Schritten das Netzwerk verbessert wird. Man kann sie nach Belieben variieren. Die besten Ergebnisse erhielt ich mit der Lernrate von 0,1.

```
# Anzahl der Knotenpunkt in jeder input, hidden und output Schicht
input_nodes = 784
hidden_nodes = 200
output_nodes = 10

#Lernrate ist 0.1
learning_rate = 0.1

# Erzeugung eines neuronalen Netzwerkes
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes,learning_rate)
```

Training des künstlichen neuronalen Netzwerkes

Nun kann man fast mit dem eigentlichen Training beginnen. Damit das zu Stande kommen kann, muss der Test-Datensatz noch eingefügt werden. Dies wird in diesen 3 Zeilen passieren.

```
#Das Laden der mnist test data file in eine Liste
test_data_file = open("Documents\Lycée 4e\Trape\mnist_dataset\mnist_test.csv",'r')
test_data_list = test_data_file.readlines()
test_data_file.close()
```

Nun muss der Test-Datensatz noch lesbar für unser Programm gemacht werden. Zuerst wird eine Schleife erzeugt, somit lernt das Netz 5 Zyklen lang - anstatt von einem. Danach werden die Datensätze aus dem Trainings-Datensatz herausgesucht, welche eine handgeschriebene Zahl beschreiben. Der Datensatz, der eine handgeschriebene Zahl beschreibt, wird nun in seine einzelnen Werte unterteilt, dies passiert durch die Funktion „record.split(',')“.

In der nächsten Zeile passiert ein wenig mehr. Es ist vorteilhaft, wenn die Eingabewerte und die Ausgabewerte im gleichen Zahlenbereich zwischen 0 und 1 sind: dem optimalen Bereich der Aktivierungsfunktion. Zuerst wird der Bereich der Farbwerte von 0 bis 255 auf 0,01 bis 1 skaliert. Der untere Bereich ist bewusst auf 0,01 gesetzt, damit verhindert wird, dass die Gewichtsaktualisierungen künstlich durch einen Nullwert vernichtet wird. Nun wird der Anfangswert durch 255 dividiert, somit kommt man in den Bereich von 0 bis 1. Danach wird der Wert mit 0,99 multipliziert und in den Bereich zwischen 0,0 und 0,99 gebracht. Zum Schluss wird noch 0,01 addiert, so kommt man schlussendlich in den Bereich von 0,01 und 1,0. Auf dem Screenshot, der direkt unter diesem Text ist, sieht man eine Liste von skalierten

Werten.

```
In [6]: for record in training_data_list:
        all_values = record.split(',')
        inputs = (numpy.asarray(all_values[1:]) / 255.0 * 0.99) + 0.01
        print(inputs)
```

```
0.88352941 0.67776471 0.99223529 0.94952941 0.76705882 0.25847059
0.01 0.01 0.01 0.01 0.01 0.01
0.01 0.01 0.01 0.01 0.01 0.20023529
0.934 0.99223529 0.99223529 0.99223529 0.99223529 0.99223529
0.99223529 0.99223529 0.99223529 0.98447059 0.37105882 0.32835294
0.32835294 0.22741176 0.16141176 0.01 0.01 0.01
0.01 0.01 0.01 0.01 0.01 0.01
0.01 0.01 0.01 0.07988235 0.86023529 0.99223529
0.99223529 0.99223529 0.99223529 0.99223529 0.77870588 0.71658824
0.96894118 0.94564706 0.01 0.01 0.01 0.01
0.01 0.01 0.01 0.01 0.01 0.01
0.01 0.01 0.01 0.01 0.01 0.01
0.01 0.01 0.32058824 0.61564706 0.42541176 0.99223529
0.99223529 0.80588235 0.05270588 0.01 0.17694118 0.60788235
0.01 0.01 0.01 0.01 0.01 0.01
0.01 0.01 0.01 0.01 0.01 0.01
0.01 0.01 0.01 0.01 0.01 0.01
0.01 0.06435294 0.01388235 0.60788235 0.99223529 0.35941176
0.01 0.01 0.01 0.01 0.01 0.01
0.01 0.01 0.01 0.01 0.01 0.01
```

In den nächsten 2 Zeilen wird beschrieben, wie das Ergebnis angeschrieben werden soll. Das richtige Ergebnis sollte im optimalen Fall einen Wert von 0,99 haben und das Falsche einen Wert von 0,01 haben. Somit wird eine Liste erstellt, in welche die Ergebnisse danach eingetragen werden. Somit sind alle Vorbereitungen für das Training abgeschlossen, nun kann man das Programm starten und trainieren lassen. Das Problem ist aber, das man kein richtiges Ergebnis bekommt und somit nicht ahnen kann, ob das KNN gut trainiert hat oder nicht.

Aber fangen wir erstmal mit dem Testen des KNNs an.

```
# epochs ist die Anzahl der Wiederholungen der Nutzung des Training Datensatzes für dieses Training
epochs = 5

for e in range(epochs):
    # durchgehen der Datensätze im Trainings Datensatz
    for record in training_data_list:
        # aufteilung der Datensätze durch ',' Kommas
        all_values = record.split(',')
        # skalieren und verschiebung der Eingaben
        inputs = (numpy.asarray(all_values[1:]) / 255.0 * 0.99) + 0.01
        # Erstellung des gewünschten Ausgabe Wertes
        targets = numpy.zeros(output_nodes) + 0.01
        # all_values[0] is der Ziel-Kennsatz für den Datensatz
        targets[int(all_values[0])] = 0.99
        n.train(inputs, targets)
    pass
pass
```


Testen des künstlichen neuronalen Netzwerkes

Das Programm, das für das Testen zuständig ist, unterscheidet sich nur wenig von dem Trainingscode. Die einzigen Unterschiede sind, dass erstens ein Variabel hinzugefügt wurde, die das korrekte Ergebnis enthält und das zweitens noch eine If-Aussage angefügt wurde, die besagt, dass eine 1 in das Scoreboard gesetzt wird, im Falle eines richtigen Ergebnisses und eine 0, falls dies nicht zutrifft. Dieses Scoreboard ist eine anfangs leere Liste, die vor dem Training erschaffen wurde.

```
# Testen des künstlichen neuronalen Netzwerkes

# Scorbboard, um zu sehen wie gut das Netz funktioniert
scorecard = []

# alle Datensätze durchgehen
for record in test_data_list:
    # teilen der Werte durch ', ' Kommas
    all_values = record.split(',')
    # richtige Anteort ist der erste Wert
    correct_label = int(all_values[0])
    # scale and shift the inputs
    inputs = (numpy.asarray(all_values[1:]) / 255.0 * 0.99) + 0.01
    # query the network
    outputs = n.query(inputs)
    # the index of the highest value corresponds to the label
    label = numpy.argmax(outputs)
    # append correct or incorrect to list
    if (label == correct_label):
        # network's answer matches correct answer, add 1 to scorecard
        scorecard.append(1)
    else:
        # network's answer doesn't match correct answer, add 0 to scorecard
        scorecard.append(0)
        pass

pass
```

Nach dem Training kann man dieses Scoreboard aufrufen. Aus dieser Liste kann man mit dem reinen Auge keine genaue Schlussfolgerung entziehen. Deshalb schrieb ich einen kleinen Code, der die Werte des Scoreboards zusammenrechnet und durch die Größe der Liste teilt, somit kommt der Prozentsatz der richtigen Ergebnisse heraus, der in meinem Fall bei 97,31% liegt. Dies ist ein ziemlich gutes Ergebnis, wenn man berücksichtigt, dass der Datensatz 60'000 handgeschriebene Zahlen enthält. Umgerechnet heißt das, dass 58'386 Zahlen erkannt wurden und dass 1'614 Werte nicht erkannt wurden.

```
In [9]: print(scorecard)
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

```
In [10]: # calculate the performance score, the fraction of correct answers  
scorecard_array = numpy.asarray(scorecard)  
print ("performance = ", scorecard_array.sum() / scorecard_array.size)
```

```
performance = 0.972
```


Schlusswort

Die Vervollständigung hat mir, wie meine vorherigen Arbeiten, sehr viel Aufschlussreiches mitgegeben. Nicht nur habe ich weiter meine Kompetenzen im Thema Informatik gestärkt, sondern ich habe mich durch solch ein aktuelles Thema ebenso auf den neusten Stand der Technik gebracht und diese neuerlernten Fähigkeiten werden mir in der Zukunft sicher noch nützlich sein.

Alles in Allem hat mir vor allem der praktische Teil enorm viel Spaß gemacht, da ich meine Informatikkenntnisse, welche ich durch meinen vorherigen „Travail personel“ erlernt habe, in einer reellen Aufgabe anwenden konnte. Dies machte mir mehr Spaß als die abstrakten, frei erfundenen Aufgaben meiner letzten Arbeit.

Auch wenn das Thema der neuronalen Netzwerke auf den ersten Blick schwierig erscheinen mag, ist es nach einigen Recherchen nicht mehr so abstrakt wie vorher. Wenn man sich erst einmal in das Thema vertieft hat, merkt man, wie interessant und weiterführend es ist.

Dieses Thema wird uns auf jeden Fall in der Zukunft noch weiter erhalten bleiben und richtungsweisend sein.

Ich bereue es jedenfalls nicht, es ausgewählt zu haben und bleibe weiter interessiert an den Neuerungen, die kommen werden.

Die Zukunft wird spannend!

Quellen

Rashid T. (2017). Neuronale Netze selbst programmieren: Ein verständlicher Einstieg mit Python. Heidelberg: O'Reilly

Nguyen C.N. und Zeigermann O. (2018). Machine Learning – kurz & gut. Heidelberg: O'Reilly

Bayer T. (2018). PC Games Hardware 10/2018. Fürth: Computec Media Group, S.100-103

Angerer C. (2018) Spektrum der Wissenschaft 3.18. Heidelberg: Spektrum der Wissenschaft Verlagsgesellschaft mbH, S.40-49

OnPulson.de (2018) Definition: Künstlich Neuronale Netze | OnPulson-Wirtschaftslexikon [online] verfügbar: <https://www.onpulson.de/lexikon/kuenstlich-neuronale-netze/> [aufgerufen: 14.10.2018]

Tawil M. (1999) Künstliche Neuronale Netze – Methoden und Anwendungen [online] verfügbar: https://www.imw.tu-clausthal.de/fileadmin/Bilder/Forschung/Publikationen/Mitt_1999/99_11.pdf [aufgerufen: 02.11.2018]

Zulkifli H. (2018) Understanding Learning Rates and How It Improves Performance in Deep Learning [online] verfügbar: <https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10> [aufgerufen: 12.01.2019]

Moeser J. (2017) Künstliche Neuronale Netze – Aufbau & Funktionsweise [online] verfügbar: <https://jaai.de/kuenstliche-neuronale-netze-aufbau-funktion-291/> [aufgerufen: 12.01.2019]

Travis E. (2017) numpy.array – NumPy v1.13 Manual [online] verfügbar: <https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.array.html> [aufgerufen: 13.01.2019]

The Scipy community (2014) scipy.special.expit - SciPy v0.14.0 Reference Guide [online] verfügbar : <https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.special.expit.html> [aufgerufen: 13.01.2019]

Günter D. () Neuronale Netze – Eine Einführung – Grundlagen [online] verfügbar: <http://www.neuronalesnetz.de/index.html> [aufgerufen: 11.03.2019]

Luber S. (2018) Was ist ein Neuronales Netz? [online] verfügbar: <https://www.bigdata-insider.de/was-ist-ein-neuronales-netz-a-686185/> [aufgerufen: 13.04.2018]

So geht Data Science (2018) Wie funktionieren künstliche neuronale Netze | Was ist ...? [online] verfügbar: <https://www.youtube.com/watch?v=wXp3BVGqMeg> [aufgerufen: 14.04.2018]

Bilderquellen:

Abbildung Deckblatt: <https://www.youtube.com/watch?v=wXp3BVGqMeg>

Abbildung 1: https://www.spieleprogrammierer.de/wiki/Datei:Nn_00.png

Abbildung 2: <https://www.youtube.com/watch?v=wXp3BVGqMeg>

Abbildung 3: <http://www.neuronalesnetz.de/matrix.html>