EINFÜHRUNG IN DIE NEURONALEN NETZWERKE

Vadim Budagov
Hochschule für Angewandte Wissenschaften Hamburg
Studiengang: European Computer Science
MatrikelNr.: 2305722

Prüfer: Prof. Dr. Michael Neitzke

Inhalt

Abbildungsverzeichnis

1 Einleitung

Im Rahmen des Studiengangs European Computer Science im vierten Semester wird eine Studienarbeit über neuronale Netzwerke erfasst. Diese soll dem Erfasser als eine Vorbereitung für das Auslandsjahr dienen. Dabei befasst sich diese Studienarbeit mit der Einführung in die neuronalen Netzwerke, als Kerngrundlage der künstlichen Intelligenz. Des Weiteren wird ein Einblick in die Funktionsweise und den Aufbau künstlicher neuronaler Netzwerke gegeben. Abschließend wird ein Versuch zur Erkennung der handschriftlichen Ziffern durchgeführt.

2 Künstliche neuronale Netzwerke

Bevor mit dem eigentlichen Versuch begonnen wird, werden hier zunächst die Begriffe *künstliche Neuronen* und *künstliche Intelligenz* erläutert. Darüber hinaus werden zwei Arten von Neuronen definiert und der Unterschied zwischen ihnen diskutiert.

Vieri Failli (2019) beschreibt eine menschliche Nervenzelle (Neuron) folgendermaßen: „Ein Neuron ist eine elektrisch erregbare Zelle, die mit Hilfe von elektrischen und chemischen Signalen Informationen aufnimmt, verarbeitet und weitergibt. Es ist eines der grundlegenden Elemente des Nervensystems.“ Dieses Prinzip der Struktur und der Funktionsweise des menschlichen und tierischen Nervensystems versucht man auf ein Computersystem abzubilden, um das System lernfähig zu machen.
Das menschliche Gehirn besitzt in jeder Hemisphäre einen primären visuellen Kortex, der 140 Millionen Neuronen mit Dutzenden von Milliarden Verbindungen enthält. Zur heutigen Zeit ist es unmöglich diese auf eine Maschine abzubilden (vgl. Nielsen 2018a: 1). Deswegen versucht man die neuronalen Netzwerke auf ein bestimmtes Problemgebiet zu spezialisieren, zum Beispiel die Sprachanalyse, die Spracherkennung oder die Gesichtserkennung.

Julian Moeser (2017) bezeichnet ein künstliches neuronales Netz als „eine Ansammlung von einzelnen Informationsverarbeitungseinheiten (Neuronen), die schichtweise in einer Netzarchitektur angeordnet sind. Im Zusammenhang mit künstlicher Intelligenz spricht man von künstlichem neuronalem Netz.“

Was man unter künstlicher Intelligenz versteht, ist im Gabler Wirtschaftslexikon folgendermaßen definiert: „Erforschung ‚intelligenten‘ Problemlösungsverfahrens sowie die Erstellung ‚intelligenter‘ Computersysteme. Künstliche Intelligenz (KI) beschäftigt sich mit Methoden, die es einem Computer ermöglichen, solche Aufgaben zu lösen, die, wenn sie vom Menschen gelöst werden, Intelligenz erfordern“.

2.1 Funktionsweise und Aufbau künstlicher neuronaler Netze

Die Neuronen (auch Knotenpunkte) eines künstlichen neuronalen Netzes sind schichtweise in sogenannten Layern angeordnet und in der Regel in einer festen Hierarchie miteinander verbunden. Die Neuronen sind dabei zumeist zwischen zwei Layern verbunden, in selteneren Fällen aber auch innerhalb eines Layers.

Zwischen den Layern oder Schichten ist jedes Neuron der einen Schicht immer mit allen Neuronen der nächsten Schicht verbunden.

Beginnend mit der Eingabeschicht (***Input Layer***) fließen Informationen über eine oder mehrere Zwischenschichten (***Hidden Layer***) bis hin zur Ausgabeschicht (***Output Layer***). Dabei ist der Output des einen Neurons der Input des nächsten. Siehe Abbildung 1 (vgl. Nielsen 2018a: 11).

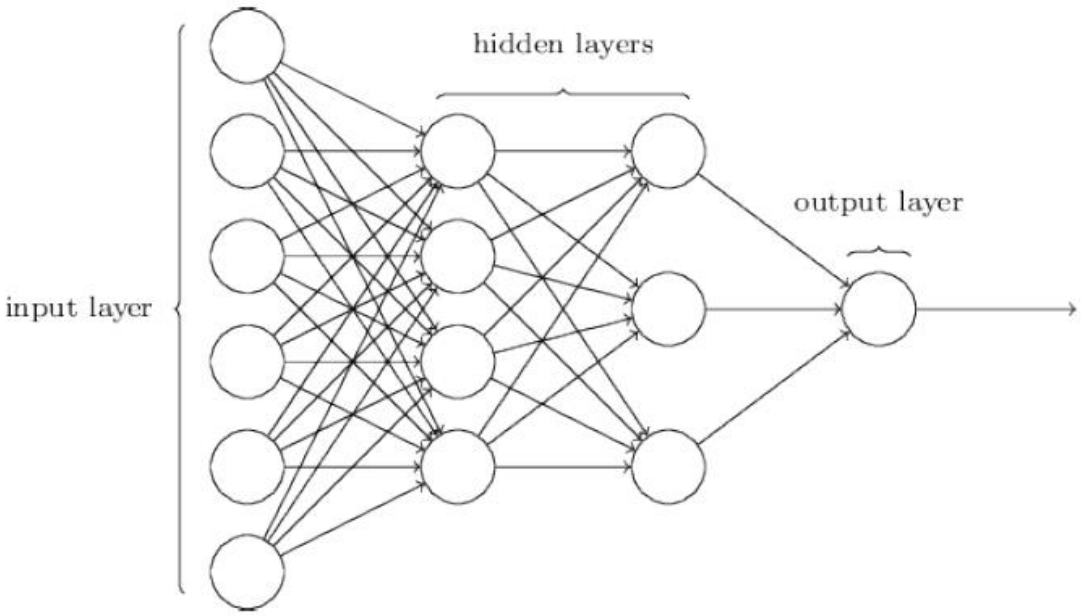


Abbildung 1: Schichten neuronaler Netze

Die **Eingabeschicht** ist der Startpunkt des Informationsflusses in einem künstlichen neuronalen Netz. Eingangssignale werden von den Neuronen am Anfang dieser Schicht aufgenommen und am Ende gewichtet an die Neuronen der ersten Zwischenschicht weitergegeben. Dabei gibt ein Neuron der Eingabeschicht die jeweilige Information an alle Neuronen der ersten Zwischenschicht weiter.

Zwischen der Eingabe- und Ausgabeschicht befindet sich in jedem künstlichen neuronalen Netz mindestens eine **Zwischenschicht**. Je mehr Zwischenschichten es gibt, desto **tiefer** ist das neuronale Netz. Im englischen spricht man daher auch von **Deep Learning**. Theoretisch ist die Anzahl der möglichen verborgenen Schichten unbegrenzt. In der Praxis bewirkt jede hinzukommende verborgene Schicht jedoch auch einen Anstieg der benötigten Rechenleistung, die für den Betrieb des Netzes notwendig ist.

Die **Ausgabeschicht** liegt hinter den Zwischenschichten und bildet die letzte Schicht in einem künstlichen neuronalen Netzwerk. In der Ausgabeschicht angeordnete Neuronen sind jeweils mit allen Neuronen der letzten Zwischenschicht verbunden. Die Ausgabeschicht stellt den Endpunkt des Informationsflusses dar und enthält das Ergebnis der Informationsverarbeitung durch das Netz (vgl. Moeser 2017).

Um die Arbeitsweise der künstlichen neuronalen Netze besser nachvollziehen zu können, werden im folgenden Abschnitt sogenannte Perzeptren betrachtet.

2.2 Perzeptron

Das Perzeptron (siehe Abbildung 2) ist ein mathematisches Modell eines künstlichen neuronalen Netzwerks. Es besteht in der einfachsten Form aus einem Neuron, dessen Ausgangsfunktion durch die Gewichtung der Eingänge und durch Schwellwerte bestimmt wird. Ein Perzeptron, das von Frank Rosenblatt im Jahr 1958 entwickelt und vorgestellt wurde, kann für maschinelles Lernen und Anwendungen der KI eingesetzt werden (vgl. Luber 2019).

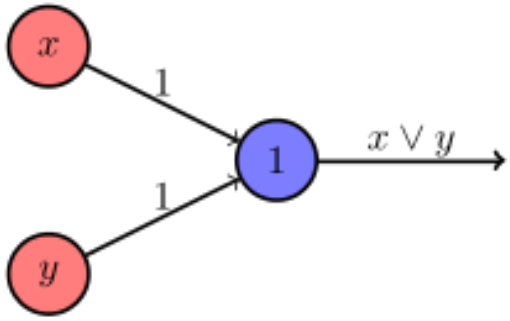


Abbildung 2: Perzeptron

Grundsätzlich ist eine Unterscheidung zwischen dem einlagigen und dem mehrlagigen Perzeptron möglich. Mehrlagige Perzeptren bestehen aus Neuronen in unterschiedlichen Lagen, die untereinander vernetzt sind. Durch das Trainieren von Perzeptren mit bestimmten Eingabemustern lassen sich anschließend ähnliche Muster in den zu analysierenden Datenmengen finden. Es handelt sich beim Trainieren des Perzeptrons um sogenanntes **überwachtes Lernen**. Die Trainingsdaten müssen hierfür zu validen Ergebnissen führen.

Ein Perzeptron besteht aus einem Neuron mit einer binären Ausgabe. Der Ausgang bzw. die Aktivierungsfunktion kann zwei Zustände annehmen: 1 -> aktiv oder 0 -> inaktiv. Ist der Ausgang aktiv, so sagt man auch „das Perzeptron oder das Neuron feuert“. Um den Zustand am Anfang zu erzeugen, besitzt das Perzeptron mehrere Eingänge. Die Eingänge haben eine bestimmte veränderbare Gewichtung. Wird ein gesetzter Schwellwert durch die Gewichtungen aller Eingänge über- oder unterschritten, verändert sich der Zustand des Neuronen Ausgangs. Durch das Trainieren eines Perzeptrons mit vorgegebenen Datenmustern verändert sich die Gewichtung der Eingänge.

Das Lernen beruht auf der Anpassung der Gewichtungen. Damit ein Perzeptron tatsächlich so trainiert werden kann, dass es valide Ergebnisse erzielt, müssen die Daten linear trennbar sein. Unter linear trennbar (engl.: **linear separabel**) versteht man die Eigenschaft, dass die in einem Diagramm dargestellten Daten durch eine Linie trennbar sind und diese Linie sie einem bestimmten Ergebnis oder Muster zuordnet (vgl. Luber 2019).

Typische Anwendungsgebiete für Perzeptren: die Analyse der Wetterdaten, die Analyse von Mess- und Sensordaten, die Analyse von Aktienkursen und Wirtschaftsdaten, die Handschrifterkennung, usw.

2.3 Lernalgorithmus von Perzeptron

In diesem Abschnitt betrachtet man, wie ein Perzeptron funktioniert und wie die Berechnungen durchgeführt werden.

Es sei gegeben:

Aktivierungsfunktion: Heavisidesche Stufenfunktion

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

Lernalgorithmus:

1) Zunächst müssen alle Gewichte initialisiert werden:
z.B.: alle $W_i = 0$ oder kleine zufällige Werte.

2) Gegeben ist ein Trainingsset (training_data) von Datenvektoren:
$\vec{x}_1 \; bis \; \vec{x}_m, \; \overrightarrow{x_i} = (x_{i1} \ldots, x_{in})$ jeweils mit erwartetem Ergebnis d_i (erwarteter Wert!)

3) Für alle $i = 1 \ldots m$ ausführen (Loop)

- Perzeptron bekommt als Input: $\overrightarrow{x_i}$ und berechnet Output y_i (berechneter Wert!)
- Jetzt werden die Gewichte aktualisiert $\boxed{w_j = \; w_j + \boldsymbol{\alpha} * x_i, j \cdot (d_i - y_i)}$, wobei $\boldsymbol{\alpha}$ **ein** Lernfaktor im Bereich von (0, 1] ist.

Wenn die Ausgabe richtig war, so wird nichts an gewichten verändert.

Die Aktualisierung von Gewichten im Loop 3) kann man auf zwei Art und Weise vornehmen. In Informatik unterscheidet man zwischen einem Offline und einem Online Algorithmus.

Online: Währen das Loop durchläuft, werden sofort für jeden Input die Gewichte aktualisiert. Dann das nächste Input und so weiter.

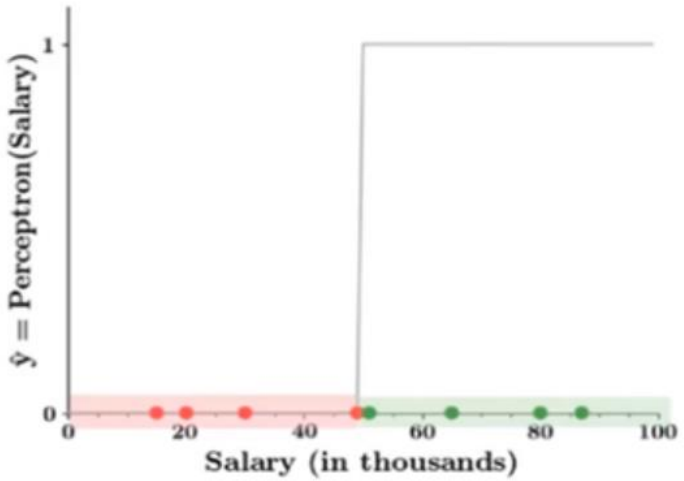
Offline: Man nimmt alle Inputs und berechnet am Ende einen globalen Fehler. Danach werden in einem Schritt für das gesamte Trainingsset Gewichte aktualisiert (vgl. Weitz 2017).

2.4 Sigmoid Neuronen

Der Baustein der tiefen neuronalen Netzwerke wird als Sigmoid Neuron bezeichnet. Die Sigmoid Neuronen sind den Perzeptronen ähnlich, jedoch sind sie so modifiziert, dass die Ausgabe von Sigmoid Neuronen viel weicher als die funktionelle Stufenfunktion von Perzeptron ist.

Aus der mathematischen Darstellung kann man sagen, dass die Schwellenwertlogik des Perzeptrons sehr hart ist. Sehen wir uns die strenge Schwellenwertlogik mit einem Beispiel an. Man betrachte den Entscheidungsfindungsprozess einer Person, ob sie ein Auto kaufen möchte oder nicht. Als Grundlage dient eine einzige Eingabe, X1 – Gehalt und die folgenden Einstellungen für den Schwellenwert b (W_o) = -10 und das Gewicht W_1 = 0,2. Die Ausgabe des Perzeptron-Modells ist in der nächsten Abbildung zu sehen. Eine Stufenfunktion (engl. Step funcion) ist zu erkennen (vgl. Kumar 2019).

Salary (in thousands)	Can buy a car?
80	1
20	0
65	1
15	0
30	0
49	0
51	1
87	1



Data (Left) & Graphical Representation of Output(Right)

Abbildung 3: Stufenfunktion

Um die Ergebnisse nachzuvollziehen, werden im Folgenden die Daten (Gewichte) in Heaviside Funktion eingesetzt und berechnet:

H(-10 + 0,2 + **15** * 0,2) = H(-6,8) = 0

H(-10 + 0,2 + **20** * 0,2) = H(-5,8) = 0

H(-10 + 0,2 + **30** * 0,2) = H(-3,8) = 0

H(-10 + 0,2 + **49** * 0,2) = H(0) = 0 //Schwelle

H(-10 + 0,2 + **51** * 0,2) = H(0,4) = 1

H(-10 + 0,2 + **65** * 0,2) = H(3,2) = 1

H(-10 + 0,2 + **80** * 0,2) = H(6,2) = 1

H(-10 + 0,2 + **87** * 0,2) = H(7,6) = 1

Im nächsten Schritt werden in der Regel die Ergebnisse mit den Testdaten verglichen und evtl. die Gewichte aktualisiert.

Im Gegensatz zu Perzeptron reagiert Sigmoid Neuron auf kleine Änderungen der Eingabe mit einer kleinen Änderung in der Ausgabe. Diese Eigenschaft wird auch als „S“-förmige Kurve bezeichnet. Die an der häufigsten verwendeten Funktion ist die Logistikfunktion. In der Abbildung 4 sieht man die Darstellung der Logistikfunktion.

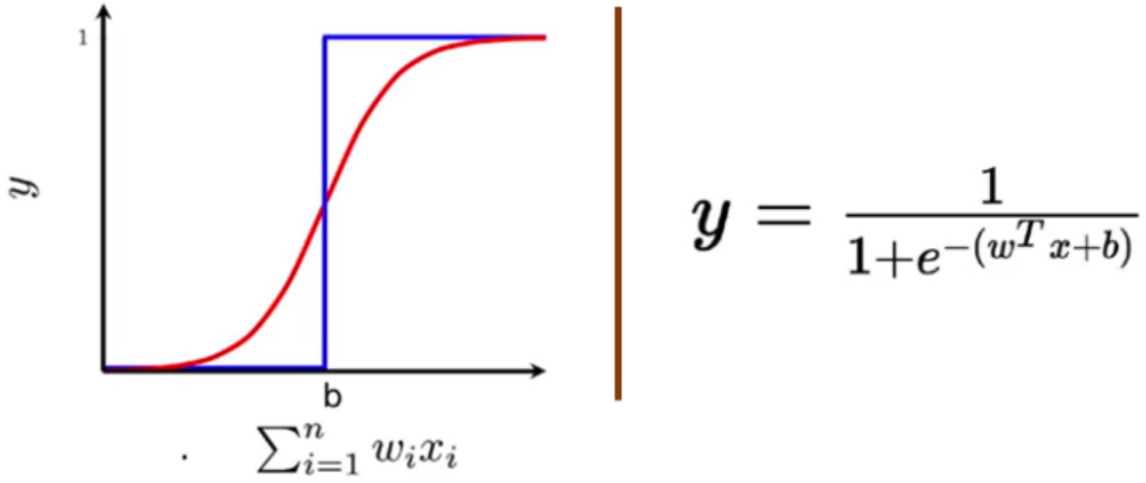


Abbildung 4: Logistikfunktion

An der Schwelle **b** sieht man keinen scharfen Übergang mehr. Die Ausgabe des Sigmoid Neurons ist nicht 0 oder 1. Stattdessen handelt es sich um einen reellen Wert zwischen 0 und 1, der als Wahrscheinlichkeit interpretiert werden kann.

3 Erkennung von handschriftlichen Ziffern

Im Folgenden wird der Versuch beschrieben und diskutiert. Dabei wird auf gewonnene Erkenntnisse eingegangen und schließlich das Ergebnis erläutert. Die Idee ist es, eine große Anzahl von handschriftlichen Ziffern, die sogenannten Trainingsbeispiele, aufzugreifen und dann ein System zu entwickeln, das aus diesen Trainingsbeispielen lernen kann. Je größer die Anzahl der Beispiele, desto mehr lernt das System davon und kann somit die Richtigkeit der Zahlen verbessern. Der Großteil der Information zu diesem Versuch wurde aus dem Buch Neuronal Networks and Deep Learning von Michael Nielsen gewonnen (vgl. Nielsen 2018a). Der vorgegebene Quellcode ist auf GitHub vom Michael Nielsen zu finden (vgl. Nielsen 2018b).

Zunächst werden alle Werkzeuge aufgelistet und erklärt, die bei der Durchführung des Versuchs eingesetzt wurden. Anschließend wird der Versuch näher erläutert.

3.1 Werkzeuge

Der Versuch wird auf dem Rechner ThinkPad E460 mit Windows 10 durchgeführt. Es ist zu beachten, dass die Ergebnisse auf unterschiedlichen Rechnern voneinander abweichen können. Das Programm ist in Python Version 2.7 geschrieben. Entscheidet man sich für eine andere Python Version, so ist davon auszugehen, dass der Code an mehreren Stellen angepasst werden muss. Beispielweise hat man bei Python 2 einen minimalen Unterschied zwischen Funktionen „range()“ und „xrange()“. Allerdings verhält sich mit Python 3 die Funktion „range()“ wie “xrange()“, weshalb die zweite Funktion überflüssig wird. Außerdem unterscheiden sich auch die funktionalen Bausteinen map(), filter() und zip() in den beiden Versionen (vgl. Grimm 2009).

Der Code wird in Eclipse Java Oxygen manipuliert und untersucht. Um den Python-Code auf Eclipse auszuführen, ist eine Installation von PyDev Bibliothek notwendig. Alle

Installationsschritte wurden von Johannes Petz anschaulich definiert und sind auf seiner Webseite zu finden (vgl. Johannes Petz 2014). Außerdem ist eine Python Bibliothek namens NumPy(Numerischen Python) notwendig, um die Funktionen der linearen Algebra einzusetzen.

3.2 Durchführung des Versuchs

Es ist ein MNIST (engl.: Modifies National Institute of Standards and Technology database) Data mit handgeschriebenen Zahlen gegeben, die für die Trainingszwecke eines neuronalen Netzwerkes eingesetzt werden. Dies besteht aus vielen 28 X 28 gescannten Pixelbildern. Anhand dieser Information kann man die Größe der Eingabeschicht definieren und zwar 784 = 28 x 28. In Abbildung 5 werden einfachheitshalber nicht alle 784 Input-Neuronen dargestellt. Die Eingabe-Pixels sind Graustufen mit dem Wert 0.0 für Weiß und 1.0 für Schwarz und die Werte dazwischen.

Die Anzahl der Neuronen in der Zwischenschicht ist als **n** definiert, da **n** im Laufe des Versuchs verschiedene Werte zugewiesen werden können. In der Abbildung 5 ist eine kleine Zwischenschicht mit n=15 zu sehen. Die Ausgabeschicht besteht aus 10 Neuronen, die für die Erkennung der Zahlen von 0 bis 9 zuständig sind.

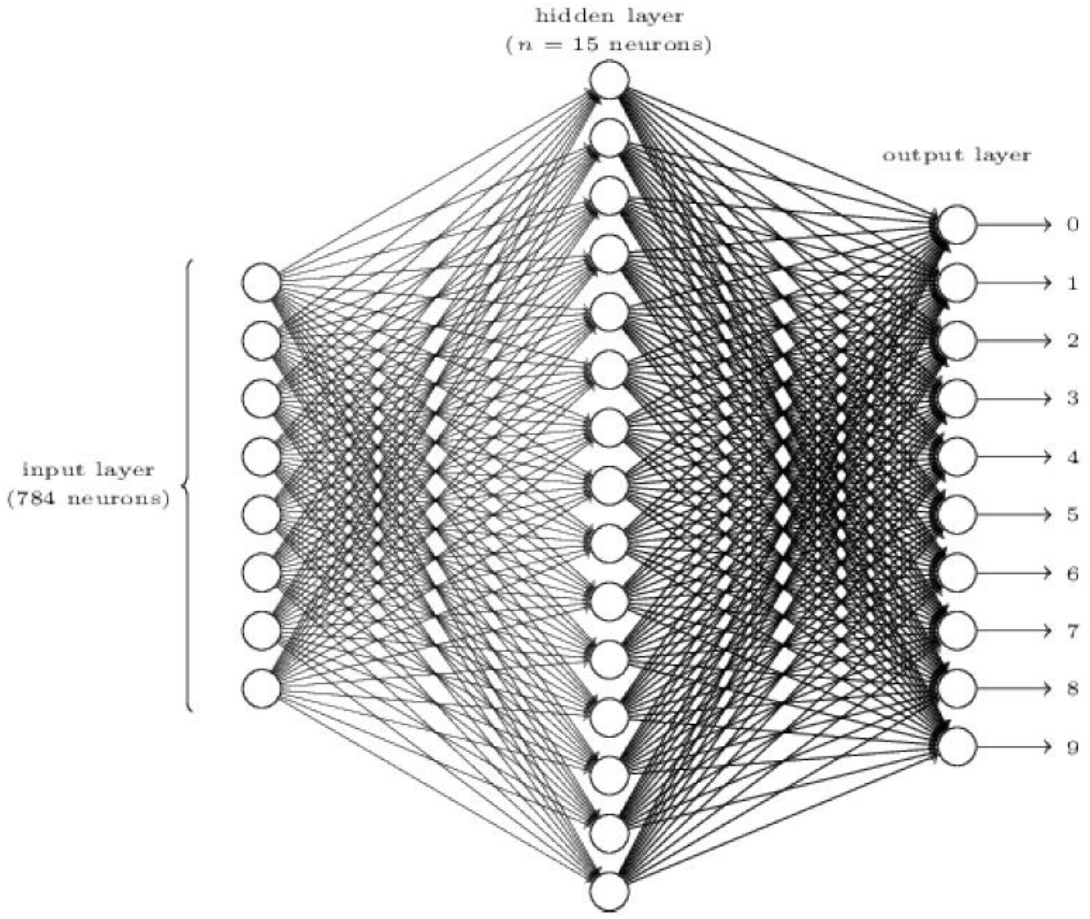


Abbildung 5: Neuronales Netz zur Erkennung von handschriftlichen Ziffern

Das Programm besteht aus drei Dateien, network.py, mnist_loader.py und mnist.pkl.gz (MNIST). Die Klasse Network aus network.py Datei ist das Herzstück des Programms. Siehe Abbildung unten.

```
class Network(object):
    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

Abbildung 6: Network Class

In diesem Codeabschnitt enthält die Liste die Anzahl der Neuronen in den jeweiligen Schichten. Zum Beispiel Network Objekt könnte folgendermaßen aussehen: net = Network([5, 10, 2]). Dies bedeutet, dass ein Network aus 5 Neuronen in der ersten Schicht, 10 Neuronen in der zweiten Schicht und 2 Neuronen in der letzten Schicht besteht.

Damit das Network in der Lage ist zu lernen, wurde eine SGD(Stochastic Gradien Descent/ deu.: Stochastischer Gradientenabstieg) Methode implementiert. Siehe Abbildung 7.

```
    def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):

        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)

```

Abbildung 7: Stochastischer Gradientenabstieg

Der Übergabeparameter training_data ist ein Tupel (x, y), die die Trainingseingaben und die gewünschten Trainingsausgaben darstellen. Der Parameter epochs definiert die Anzahl von Trainings-Epochen. Die mini_batch_size definiert eine Größe, bei der die Stichproben verwendet werden sollen. eta ist die Lernrate.

Wenn das optionale Argument test_data angegeben wird, wertet das Programm das Netzwerk nach jeder Trainingseinheit aus und gibt einen Teil des Fortschritts aus. Außerdem wird für jedes mini_batch Gradientenabstieg angewendet, um die Gewichte zu aktualisieren. Dies geschieht durch self.update_mini_batch(mini_batch, eta).

Aus der Menge der zur Verfügung stehenden Trainingsdaten nimmt man also nur eine Stichprobe von mini_batch Datensätzen, führt damit ein Training, also eine Lernphase durch. Danach nimmt man die nächste Stichprobe von mini_batch Datensätzen bis alle Trainingsdaten verwendet wurden. Dies ist dann eine Trainings-Epoche, auf die eine weitere Epoche folgen kann.

Der Versuch wird in einem Python Shell ausgeführt. Zunächst soll die MNIST Data geladen werden. Dies macht das kleine Hilfsprogramm mnist_loader.py, welches importiert werden muss:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
>>>
```

Abbildung 8: import mnist_loader

Nachdem die MNIST Datei geladen wurde, wird die Network Datei importiert, um das neuronale Netzwerk zu definieren. D.h.: dessen Schichten mit der gewünschten Anzahl von Neuronen.

```
>>> import network
>>> net = network.Network([784, 30, 10])
>>>
```

Abbildung 9: import network

Es wurde ein Objekt net vom Typ Network erstellt. Mit Eingabeschicht = 784, Zwischenschicht = 30 und Ausgabeschicht = 10. Schließlich ruft man die SGD Methode, um von MNIST Datei zu lernen. Man definiere die geeigneten Hyper-Parameter mit: 30 Epochen, 10 mini_batch und 3.0 für die Lernrate.

```
>>> net.SGD(training_data, 30, 10, 3.0, test_data = test_data)
```

Abbildung 10: SGD Ausführung

Als Ergebnis erhält man die Klassifizierungsrate von trainiertem neuronalem Netzwerk in Prozent ausgegeben. Dies beträgt in diesem Fall 95,22% bei Epoche 28.

```
Epoch 21: 9491 / 10000
Epoch 22: 9511 / 10000
Epoch 23: 9502 / 10000
Epoch 24: 9498 / 10000
Epoch 25: 9496 / 10000
Epoch 26: 9493 / 10000
Epoch 27: 9488 / 10000
Epoch 28: 9522 / 10000
Epoch 29: 9492 / 10000
>>>
```

Abbildung 11: Ausgabe mit 30 Zwischenschichten

Um die besseren Ergebnisse zu erzielen, wird die Anzahl von Neuronen in der Zwischenschicht von 30 auf 100 erhöht.

```
>>> net = network.Network([784, 100, 10])
>>> net.SGD(training_data, 30, 10, 3.0, test_data = test_data)
```

Abbildung 12: Network mit 100 Zwischenschichten

Obwohl die Ausführung aufgrund größerer Zwischenschicht länger dauert, liefert dieser Versuch bessere Ergebnisse. Siehe Epoch 23. Die Testdaten wurden mit 96,85% korrekt klassifiziert.

```
Epoch 22: 9655 / 10000
Epoch 23: 9685 / 10000
Epoch 24: 9670 / 10000
Epoch 25: 9663 / 10000
Epoch 26: 9665 / 10000
Epoch 27: 9668 / 10000
Epoch 28: 9667 / 10000
Epoch 29: 9646 / 10000
>>>
```

Abbildung 13: Ausgabe mit 100 Zwischenschichten

Außerdem wurde mit Hyper-Parametern experementiert und folgende Erkenntnisse gewonnen: Je höher die Lernrate des neuronalen Netzwerkes, desto höher die Ergebnisse und umgekehrt. Zum Beispiel bei einer Lernrate von 0.001 hat man 21,20% und bei einer Lernrate von 100 über 100%. Der Auswahl von geeigneten Lernrate ist essenziell, um gute Ergebnisse zu erzielen. Man kann aber auch die Zwischenschicht weglassen und nur mit Eingabe- und Ausgabeschichten arbeiten. Dies verbessert nicht die Klassifikationsrate, sondern nur die Geschwindigkeit des Lernens.

4 Schlussfolgerung

In dieser Studienarbeit wurden die wesentlichen Kenntnisse, wie der Grundaufbau und die Funktionsweise der neuronalen Netzwerke, erworben. Diese wurden am Beispiel von Perzeptren und dessen Lernalgorithmus erklärt und dargestellt. Außerdem wurden Perzeptren den Sigmoid Neuronen gegenübergestellt und der Unterschied zwischen ihnen diskutiert.

Des Weiteren wurde mit einem praxisbezogenen Beispiel, der Erkennung von handschriftlichen Ziffern, experimentiert. Dadurch wird deutlich, wie wichtig es ist, den Aufbau von Schichten eines neuronalen Netzwerkes zu verstehen. Im Laufe des Versuchs hat sich herausgestellt, dass die Anzahl der Neuronen in den jeweiligen Schichten und die geeigneten Hyper-Parameter ausschlaggebend sind, um optimale Ergebnisse zu erzielen. Dies kann die Lernrate eines neuronalen Netzwerkes verbessern.

Die künstliche Intelligenz ist ein umfangreiches Themengebiet, welches noch weiterer intensiver Forschung bedarf. Heutzutage sind neuronale Netzwerke auf dem Vormarsch und werden in der Zukunft eine immer größere Rolle spielen. Arbeitsabläufe, die algorithmisiert werden können, werden mit der Zeit durch die künstliche Intelligenz ersetz.

5 Quellenangabe

Failli, Vieri (2019): Wie funktioniert ein Neuron? https://www.wingsforlife.com/de/aktuelles/wie-funktioniert-ein-neuron-561/ (Zugegriffen am: 13.05.2019).

Grimm, Rainer (2009): Umstieg auf Python 3. https://www.linux-magazin.de/ausgaben/2009/09/im-zeichen-der-drei/ (Zugegriffen am: 13.05.2019).

Kumar, Niranjan (2019): Sigmoid Neuron – Building Block of Deep Neuronal Networks. https://towardsdatascience.com/sigmoid-neuron-deep-neural-networks-a4cd35b629d7 (Zugegriffen am 13.05.2019).

Luber, Stefan (2019): Was ist ein Perzeptron?. https://www.bigdata-insider.de/was-ist-ein-perzeptron-a-798367/ (Zugegriffen am: 13.05.2019).

Moeser, Julian (2017): Künstliche Neuronale Netze – Aufbau & Funktionsweise. https://jaai.de/kuenstliche-neuronale-netze-aufbau-funktion-291/(Zugegriffen am: 13.05.2019).

Nielsen, Michael (2018a): Neural Network and Deep Learning. http://neuralnetworksanddeeplearning.com (Zugegriffen am: 13.05.2019).

Nielsen, Michael (2018b): Neural Network and Deep Learning. https://github.com/mnielsen/neural-networks-and-deep-learning (Zugegriffen am: 13.05.2019).

Petz, Johannes (2014): PyDev in Eclipse integrieren. https://www.johannespetz.de/eclipse-als-python-entwicklungsumgebung/ (Zugegriffen am: 13.05.2019).

Weitz, Edmund (2017): Das Perzeptron. https://www.youtube.com/watch?v=RyHQPlJWLjA (Zugegriffen am: 13.05.2019).