Funktionsweise künstlicher neuronaler Netzwerke: Grundlagen und Programmierung eines Beispiels

Elias Krainer
8. Klasse

Betreuer: Mag. Gernot Schatzdorfer
Modellschule Graz, Fröbelgasse 28, 8010 Graz
Februar 2020

Abstract

Künstliche neuronale Netzwerke sind eine mathematische Nachbildung der menschlichen beziehungsweise tierischen Reizverarbeitung im Nervensystem. Sie sind selbstlernende Algorithmen und dadurch ein Zweig der künstlichen Intelligenz, ein Teilbereich der Informatik. Sie haben eine Vielzahl von Anwendungsgebieten. Diese reichen von der Gesichtserkennung bis zur Fahrzeugsteuerung.

Die vorliegende Arbeit gibt einen Einblick in den Aufbau und die Funktionsweise von künstlichen neuronalen Netzen. Anfangs wird der Algorithmus theoretisch beleuchtet, das künstliche neuronale Netzwerk wird aus dem biologischen Vorbild modelliert und mathematisch, in Bezug auf Aufbau und Verwendung, betrachtet. Danach wird ein vielverwendetes Lernverfahren, das Gradientenabstiegsverfahren, hergeleitet. In einem weiteren Teil dieser Arbeit wird das künstliche neuronale Netz von der praktischen Seite beschrieben, indem es in ein Computersystem implementiert wird. Das Programm lernt erfolgreich, handgeschriebene Ziffern selbstständig zu klassifizieren. Abschließend werden gesellschaftliche und ethische Probleme, welche mit der Verwendung künstlicher neuronaler Netze einher gehen, untersucht.

Inhalt

1. Einleitung

Ein Computer ist eine automatisierte Rechenmaschine. Er kann innerhalb von Millisekunden vielstellige Zahlen multiplizieren. Im Rechnen und allen Aufgaben, welche sich durch Rechenoperationen zusammensetzen lassen, ist er uns Menschen unbestreitbar überlegen. Doch gibt es eine Vielzahl von Problemstellungen, in welchen uns ein Computer nur schwer bis gar nicht das Wasser reichen kann. Diese Aufgaben reichen von der Gesichtserkennung bis zur Fahrzeugsteuerung.

Um Computern diese Art der Aufgabenstellungen zugänglich zu machen, wurden künstliche neuronale Netze entwickelt. Diese sind eine mathematische Nachbildung der neuronalen Reizverarbeitung im Gehirn. Auch wenn sie ein Teilgebiet der künstlichen Intelligenz sind, muss hier klar zu „Intelligenz“ im allgemeinen Sinn differenziert werden. Hier geht es nicht um denkende Maschinen, welche womöglich sogar Gefühle entwickeln. Nein, künstliche neuronale Netzwerke sind rein selbständig lernende Algorithmen.

Sie sind ein Werkzeug für eine unglaubliche Vielzahl und Diversität an Anwendungsgebieten. Sie suchen aus, welche Werbung geschaltet wird und handeln an der Börse. Sie werden in der Überwachung sowie für die Steuerung autonomer Fahrzeuge verwendet.

Das Ziel dieser Arbeit ist, die Funktionsweise von künstlichen neuronalen Netzwerken zu beschreiben. Diese werden einerseits von ihrer theoretischen Seite beleuchtet, so befasst sich ein wesentlicher Teil dieser Arbeit mit der Modellbildung von biologischen Neuronen, welche im folgendem optimiert und zu einem neuronalen Netz zusammengefügt werden, sowie mit dem Lernverfahren dieser Netze. Andererseits wird die praktische Seite eines künstlichen neuronalen Netzes anhand eines Beispiels dargelegt. Im Rahmen dieser Arbeit wird ein Netz lernen, selbstständig handgeschriebene Ziffern zu klassifizieren. Abschließend beschäftigt sich diese Arbeit mit gesellschaftlichen und ethischen Problemen, welche mit der Verwendung dieser Algorithmen einhergehen.

2. Das Neuron

2.1. Das biologische Neuron

Künstliche neuronale Netze sind dem Gehirn von Mensch und Tier nachempfunden. Ihr biologisches Vorbild definiert sich durch ein Netzwerk aus Nervenzellen, den Neuronen. Allein ist ein Neuron lediglich ein simples Rechenelement, doch den Netzwerken, welche sie bilden, wird der Grund für die menschliche Intelligenz zugesprochen (vgl. Kinnebrock, 1992, S. 11).

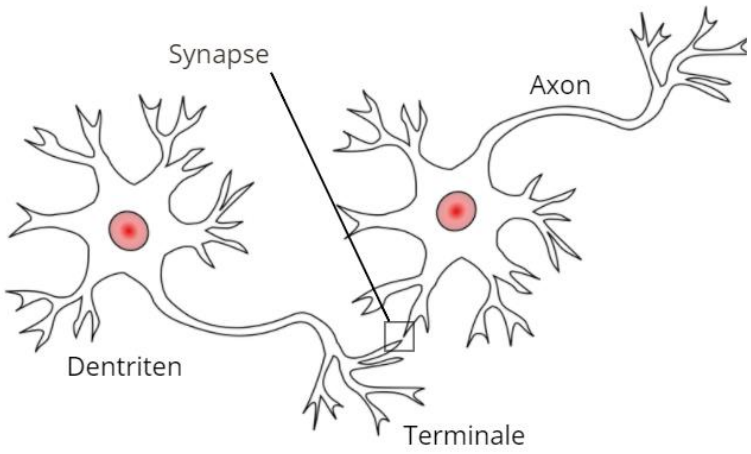

Abbildung 1: biologische Neuronen, Quelle: https://de.wikipedia.org/wiki/Datei:Neurons_uni_bi_multi_pseudouni.svg, CC BY-SA 3.0, in Inkscape bearbeitet

Es gibt zwar verschiedene Arten von Neuronen, doch sie funktionieren alle nach demselben Prinzip: Elektrische Signale (Reize) werden ausgehend von Rezeptoren (biologische Sensoren) oder anderen Neuronen über die Dendriten in das Neuron geleitet. Zwischen den Dendriten des Neurons und den Axonterminalen der zuvor geschalteten Nervenzellen befindet sich ein kleiner Spalt, über welchen die Signale chemisch übertragen werden. Diese Verbindung wird als Synapse bezeichnet. Sie hemmt oder stärkt eingehende Signale. Da die Wirkung der Synapse auf eintreffende Reize veränderbar ist, ist die Synapse ist bedeutend für das Lernverhalten eines neuronalen Netzes (vgl. Kinnebrock, 1992, S. 15).

Das Neuron sammelt alle eintreffenden Reize in einem gemeinsamen Speicher. Übersteigen diese einen Schwellenwert, schickt es über seine Terminale ein Signal an die folgenden Neuronen. Biologen sagen dazu: „das Neuron feuert“

2.2. Mathematische Umsetzung

Die Modellierung eines neuronalen Netzes hilft dabei, die menschliche Informationsverarbeitung auf Computern nachzubilden. Allererst werden die einzelnen Elemente eines neuronalen Netzes mathematisch nachgebildet: Die Neuronen.

In dem mathematischen Modell werden Reize (Signale) durch reelle Zahlenwerte dargestellt (vgl. Rashid, 2017, S. 32). Somit besteht der Eingang des Neurons aus mehreren Zahlenwerten a, welche entweder von der Umgebung (dem Input) oder von vorgeschalteten Neuronen kommen.

Die Synapsen, welche die Eingangssignale verstärken bzw. schwächen werden ebenfalls durch reelle Zahlenwerte modelliert, den Gewichten w (vgl. Kinnebrock, 1992, S. 15). Diese werden mit den eingehenden Signalen multipliziert. Daraus folgt, dass ein großes Gewicht das eingehende Signal vergrößert (stärkt) und ein kleines Gewicht das eingehende Signal verkleinert (schwächt), wobei ein Gewicht natürlich auch negativ sein kann.

Bei diesem mathematischen Modell eines neuronalen Netzes wird davon ausgegangen, dass alle eingehenden Signale zur selben Zeit eintreffen. Dies erlaubt, den vorhin erwähnten Speicher, welcher alle eingehende Reize zusammen speichert, als Summe der gewichteten Eingangssignale zu schreiben (vgl. Kinnebrock, 1992, S. 16). Also werden alle eingehenden Signale, nachdem sie durch die Synapsen verstärkt oder geschwächt werden, summiert.

Das momentane Modell lässt sich in folgender Abbildung übersichtlich ablesen, wobei y für die Ausgabe des Neurons steht.

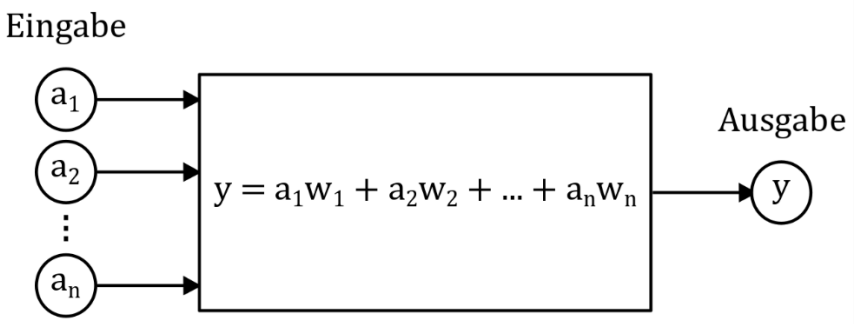

Abbildung 2: Vorläufiges Modell einer Nervenzelle, in Inkscape erstellt

Dieses lässt sich durch die Summenfunktion verallgemeinern, wobei n für die Anzahl der Eingangssignale steht:

$$y = \sum_{i=1}^{n} a_i w_i$$

Der Schwellenwert wird mit einer reellen Zahl beschrieben, welcher von der gewichteten Summe der Eingangssignale subtrahiert wird. Somit ist der Output y des Neurons erst dann positiv, wenn die eintreffenden Signale größer als der Schwellenwert sind. Jedoch spricht man anstatt vom Schwellenwert meist vom Bias b, welcher das Negative vom Schwellenwert ist (vgl. Ertel, 2008, S: 244). Also sieht die Formel, welche ein Neuron simuliert, so aus:

$$y = \sum_{i=1}^{n} a_i \, w_i + b$$

Die Formel lässt sich auch mithilfe von Vektoren nachbilden, wobei die Summe als Skalarprodukt eines Vektors aller Eingangssignale $\vec{a}$ und eines Vektors aller Gewichte $\vec{w}$ geschrieben wird. Die Beschreibung eines Neurons durch Vektoren wird bei der Programmierung noch nützlich sein, da die Eingaben meist in vektorähnlicher Form vorliegen.

$$\vec{a} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix} \quad \vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{pmatrix} \quad \vec{a} \cdot \vec{w} = a_1 w_1 + a_2 w_2 + a_3 w_3 + \cdots + a_n w_n$$

$$y = \vec{a} \cdot \vec{w} + b$$

Diese Formel entspricht der Gleichung einer Geraden mit der Steigung w und der Verschiebung b, bei einer einzelnen Eingabe. Gibt es mehrere Eingaben, wird das Neuron zur (Hyper-) Ebene (vgl. Alpaydin, 2019, S. 280). Was diese über die Funktion eines Neurons aussagt, wird im nächsten Unterkapitel geklärt.

Diese Formel kann weiter vereinfacht werden, indem der Bias zum Skalarprodukt hinzugefügt wird. So wird der Bias zum nullten Element des Gewichtsvektors und eine Eins am Anfang des Vektors der Eingangssignale angehängt. Dies erspart eine separate Behandlung des Bias für spätere Rechenschritte.

$$\vec{a} = \begin{pmatrix} 1 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix} \quad \vec{w} = \begin{pmatrix} b \\ w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{pmatrix} \quad \vec{a} \cdot \vec{w} = 1b + a_1 w_1 + a_2 w_2 + \cdots + a_n w_n$$

$$y = \vec{a} \cdot \vec{w}$$

Wie in einem späteren Kapitel noch geklärt wird, darf ein Neuron für mehrlagige neuronale Netze nicht linear sein. Deshalb verwendet man meist eine sogenannte Aktivierungsfunktion, welche die bisherige Formel umschließt. Im Falle einzelner Neuronen wird sie meist verwendet, um die Ausgabe zu formatieren. Hier wird sie mit f bezeichnet.

$$y = f(\vec{a} \cdot \vec{w})$$

Ein weit verbreitetes Beispiel für solch eine Aktivierungsfunktion ist die Stufenfunktion. Diese gibt für alle negativen Werte des Neurons 0 aus und für alle positiven 1 (vgl. Rashid, 2017, S. 33).

$$s(x) = \begin{cases} 1; \text{wenn } x > 0 \\ 0; \text{andernfalls} \end{cases}$$

In folgender Grafik wird abschließend das Verhalten eines Neurons bei drei Eingaben übersichtlich zusammengefasst:

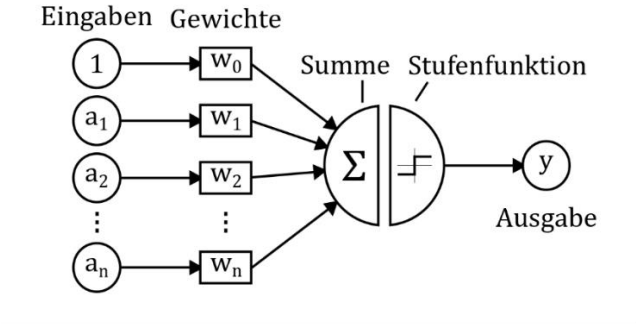

Abbildung 3: mathematisches Neuron, in Inkscape erstellt.

2.3. Das Neuron als Klassifizierer

Die Anwendung mathematischer Neuronen besteht meistens in der Lösung sogenannter Klassifizierungsprobleme (vgl. Rashid, 2017, S. 10). Bei Klassifizierungsproblemen handelt es sich um die Unterscheidung verschiedener Klassen. In diesem Zusammenhang wird eine Klasse als Zusammenfassung diverser Objekte bezeichnet (vgl. Alpaydin, 2019, S. 24). Zum Beispiel lässt sich eine Klasse Marienkäfer definieren, welche alle Marienkäfer trotz

unterschiedlicher Größe und Aussehen zusammenfasst. Klassifizierungsprobleme können so einfach sein wie Raupen und Marienkäfer voneinander zu unterscheiden, aber auch die Erkennung handgeschriebener Ziffern oder unterschiedlicher Gesichter können als Klassifizierungsprobleme beschrieben werden.

Um die Rolle von Neuronen in Klassifizierungsproblemen besser verstehen zu können, wird folgendes Beispiel angeführt: Angenommen, es gibt einen Roboter, dessen Aufgabe es ist, Raupen von Marienkäfern zu unterscheiden, wobei der Roboter lediglich die Breite und die Länge der gefragten Insekten messen kann. Des Weiteren wird vorausgesetzt, dass Raupen sehr lang und schmal sind während Marienkäfer sehr kurz und breit sind. Im folgenden Diagramm sind unterschiedliche Raupen und Marienkäfer nach ihrer Länge und Breite eingezeichnet. Natürlich sind die Maße der Verständlichkeit halber ein wenig verzerrt dargestellt.

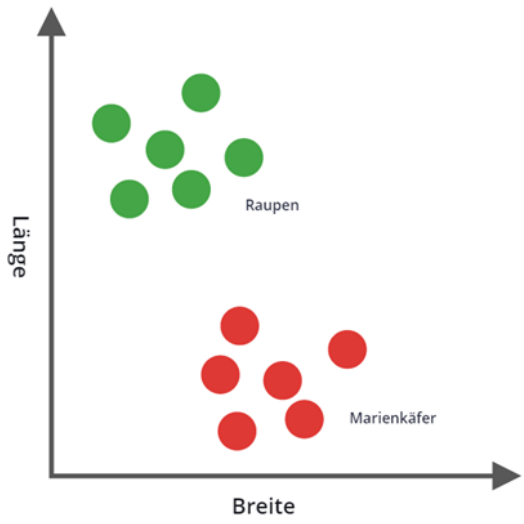

Abbildung 4: Raupen und Marienkäfer, in Inkscape erstellt

Im Folgenden wird die Länge und Breite der Insekten als die Eingaben eines Neurons betrachtet. Da zum besseren Verständnis die Aktivierungsfunktion weggelassen wird, definiert das Neuron, wie bereits erwähnt, eine Hyperebene. Eine Ebene teilt den Eingaberaum in zwei Teile. Jenen, bei dem das Neuron ein positives Ergebnis erzeugt, sowie den Teil, für den das Neuron negativ ist. In folgender Abbildung ist dieser Sachverhalt visualisiert.

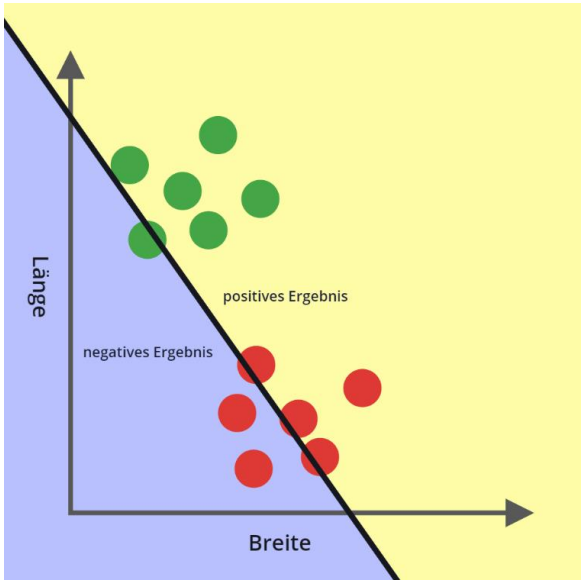

Abbildung 5: Länge und Breite als Eingabe eines Neurons, in Inkscape erstellt

Durch geschickte Anpassung der Gewichte sowie des Bias lässt sich die durch das Neuron gebildete Grenzgerade so einstellen, dass sie die Größenverhältnisse der Marienkäfer von jenen der Raupe trennt. Somit erzeugt das Neuron beispielsweise für alle Raupen ein positives und für alle Marienkäfer ein negatives Ergebnis. Natürlich sollte der Computer schlussendlich selbst diese Gerade anpassen, jedoch wird das Lernverfahren erst in einem späteren Kapitel behandelt.

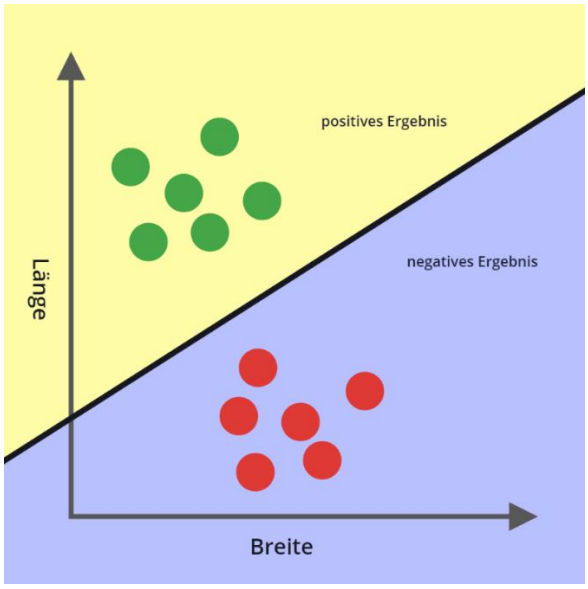

Abbildung 6: Raupen und Marienkäfer durch ein Neuron klassifiziert, in Inkscape erstellt

Ist die Unterscheidung zwischen positiv und negativ immer noch zu unspezifisch, lässt sich das Ergebnis durch eine Aktivierungsfunktion spezifizieren. Bei Verwendung der Stufenfunktion werden die Raupen als Eins und die Marienkäfer als Null gekennzeichnet.

Möchte man jedoch die Wahrscheinlichkeit haben, dass ein Insekt mit einem gewissen Größenverhältnis für eine Raupe steht, verwendet man als Aktivierungsfunktion die Sigmoidfunktion, welche eine geglättete Version der Stufenfunktion ist (vgl. Alpaydin, 2019, S. 281). Diese Funktion hat auch noch einen weiteren Vorteil gegenüber der Stufenfunktion: Sie ist differenzierbar, das ist für den Lernprozess essenziell.

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

Abbildung 7: Sigmoid und Stufenfunktion, in Geogebra erstellt

Kernideen:

- Neuronale Netze bestehen aus miteinander verschalteten Neuronen.
- Künstliche Neuronen werden über eine Hyperebene definiert, welche durch eine Aktivierungsfunktion „verzerrt“ wird.
- Ein einzelnes Neuron kann als lineare Trennebene genutzt werden, um zwei Klassen voneinander zu trennen.

3. Das künstliche neuronale Netz

3.1. Parallele Neuronen

3.1.1. Grundlagen

Parallele Neuronen werden auch als ein einlagiges künstliches neuronales Netz bezeichnet. Das sind mehrere Neuronen, welche den Input parallel und unabhängig voneinander verarbeiten. (vgl. Alpaydin, 2019, S. 281) Folgende Grafik verdeutlicht die parallele Datenverarbeitung der Neuronen.

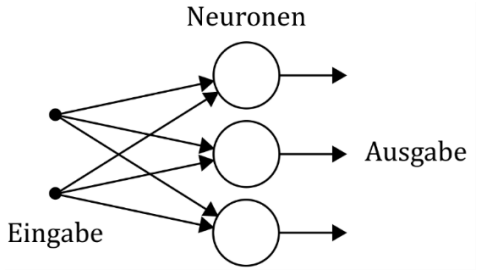

Abbildung 8: Parallele Neuronen, in Inkscape erstellt

Ihre Anwendung besteht in Klassifizierungsproblemen mit mehr als zwei Klassen. Dann wird für jede Klasse ein Neuron verwendet, welches die Objekte der jeweiligen Klasse von denen der anderen trennt (vgl. Alpaydin, 2019, S. 281). So ist zum Beispiel das Neuron, welches einer Klasse zugehörig ist, bei Objekten anderer Klassen immer negativ. Der beschriebene Sachverhalt ist auf folgendem Diagramm visualisiert.

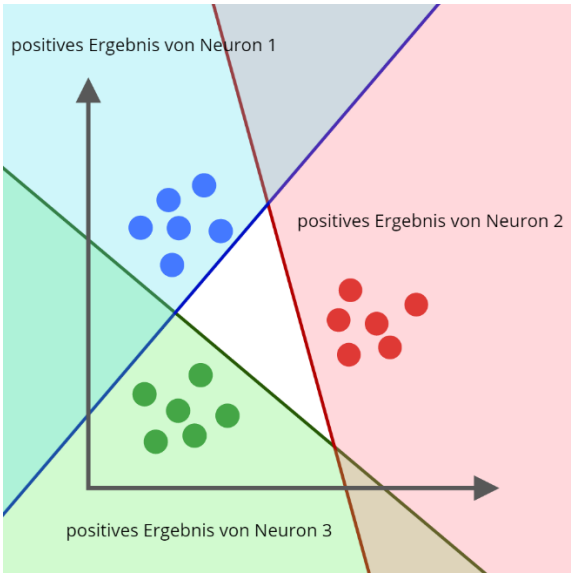

Abbildung 9: Klassifizierung bei drei Klassen, in Inkscape erstellt

3.1.2. Mathematische Definition

Jedes dieser Neuronen liefert als Output eine explizite Zahl. Diese Zahlen können für eine einfachere Weiterverarbeitung als Vektor zusammengefasst werden. So wäre es doch hilfreich, eine einzige Berechnungsformel für diese Reihe an parallelen Neuronen zu haben, welche gleich einen Vektor ausgibt. Diese ist im folgendem dargestellt, wobei m die Anzahl der parallelen Neuronen ist.

$$\vec{y} = \begin{pmatrix} \mathrm{f}(\vec{a} \cdot \overrightarrow{w_1}) \\ \mathrm{f}(\vec{a} \cdot \overrightarrow{w_2}) \\ \mathrm{f}(\vec{a} \cdot \overrightarrow{w_3}) \\ \vdots \\ \mathrm{f}(\vec{a} \cdot \vec{w}_m) \end{pmatrix}$$

Diese Formel lässt sich weiter vereinfachen. Allererst wird die Aktivierungsfunktion herausgehoben.

$$\vec{y} = \mathrm{f}\left(\begin{pmatrix} \vec{a} \cdot \overrightarrow{w_1} \\ \vec{a} \cdot \overrightarrow{w_2} \\ \vec{a} \cdot \overrightarrow{w_3} \\ \vdots \\ \vec{a} \cdot \vec{w}_m \end{pmatrix}\right)$$

Das kann man auch als Matrix-Vektor-Multiplikation schreiben. Weiterführende Informationen zur Matrix-Vektor-Multiplikation beziehungsweise Matrizen finden Sie im Anhang.

$$\vec{y} = \mathrm{f}(\mathrm{W} \cdot \vec{a})$$

3.1.3. Grenzen einlagiger neuronaler Netze

Einfache sowie parallele Neuronen haben den erheblichen Nachteil, dass sie ausschließlich linear trennbare Probleme lösen können. Somit wären folgende Verteilungen nicht eindeutig lösbar.

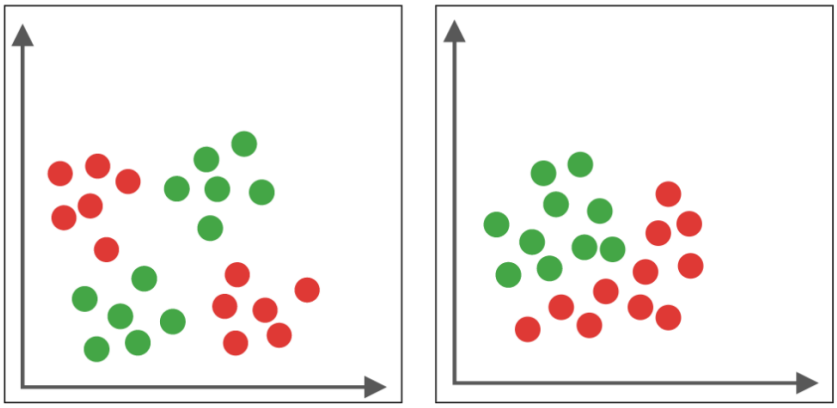

Abbildung 10: Liner unklassifizierbare Klassen, in Inkscape erstellt

Dieses Problem lässt sich beheben, indem das Netz um mehrere Lagen erweitert wird (vgl. Alpaydin, 2019, S. 288). Somit werden die Ausgaben der ersten Schicht zu den Eingaben der zweiten Schicht. Wie dies im Detail aussieht, wird im folgendem Unterkapitel geklärt.

3.2. Mehrlagige Neuronen

3.2.1. Grundlagen

Mehrlagige künstliche neuronale Netzwerke sind, wie bereits im vorigen Unterkapitel angeschnitten, mehrere hintereinandergeschaltete Schichten paralleler Neuronen, welche verknüpft werden, indem der Output der vorgeschalteten Schicht zum Input der nächsten Schicht wird (vgl. Alpaydin, 2019, S. 288). Die zusätzlichen Schichten werden verwendet, um nichtlineare Trennflächen zu schaffen. Somit sind einlagig unlösbare Probleme durch mehrlagige neuronale Netze ohne Weiteres zu lösen. Ein Beispiel für solch ein mehrlagiges neuronales Netz wird in folgender Abbildung visualisiert.

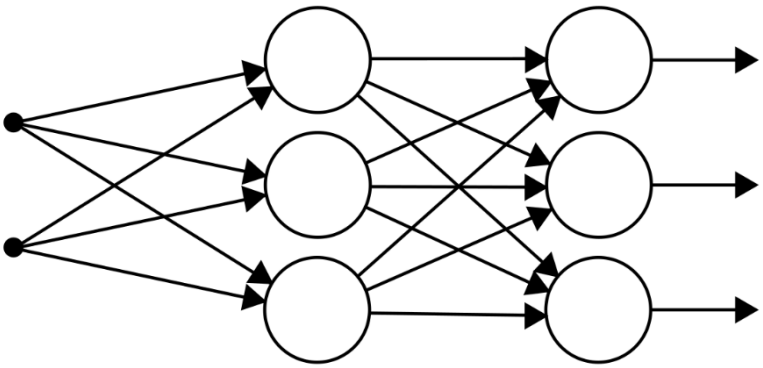

Abbildung 11: Mehrlagiges neuronales Netz, in Inkscape erstellt

Bei Verwendung der bisherigen Formeln für die einzelnen Schichten des Netzwerkes lässt sich das oben gezeigte Netz auch folgendermaßen beschreiben, wobei die tiefgestellten Zahlen für die Schicht stehen, der die Parameter zugehörig sind:

$$\vec{a}_2 = f(W_1 \cdot \vec{a}_1)$$

$$\vec{a}_3 = f(W_2 \cdot \vec{a}_2)$$

$$\vec{a}_4 = f(W_3 \cdot \vec{a}_3)$$

3.2.2. Wieso ist eine Aktivierungsfunktion bei mehreren Lagen zwingend notwendig?

Angenommen es gäbe keine solche Funktion. Ein Neuron ohne Aktivierungsfunktion macht nichts anderes als eine gewichtete Summe der Eingangswerte. Ist das Neuron in zweiter Schicht, verhalten sich die Eingaben, die es von den vorgeschalteten Neuronen bekommt, linear zum Input des Netzes. Die gewichtete Summe, die das Neuron macht, ist jedoch nicht in der Lage diese Linearität zu brechen. Somit verhält sich die Ausgabe des Neurons in zweiter Schicht wiederum wie eine Hyperebene. Damit wäre die weitere Schicht obsolet, da sowieso schon alle möglichen Hyperebenen durch eine Schicht beschrieben werden können.

3.3. Der Fehler

Der Fehler E ist eine Zahl, welche darüber Auskunft gibt, wie inkorrekt ein neuronales Netz eine Aufgabe löst (vgl. Kinnebrock, 1992, S. 40). Je kleiner der Fehler ist, desto richtiger hat das Netz das Problem gelöst. Der Fehler wird durch die Fehlerfunktion $C(a_L)_t$ beschrieben. Diese vergleicht die Ausgabe des Netzes a_L mit ihren Soll-Werten t.

$$E = C(a_L)_t$$

Als Fehlerfunktion wird meist die Summe aus $\frac{1}{2}(\text{ist} - \text{soll})^2$ verwendet (vgl. Rashid, 2017, S. 79). Diese Funktion ist stetig und leicht abzuleiten, was für den Lernalgorithmus von Vorteil ist.

$$E = \sum_i \frac{1}{2}(a_i^L - t_i)^2$$

Kernideen:

- Parallele Neuronen werden verwendet, um mehr als zwei Klassen voneinander zu trennen.
- Mehrlagige neuronale Netze können auch nicht-lineare Trennflächen erzeugen und somit auch Klassifizierungsprobleme lösen, welche linear nicht eindeutig lösbar sind.
- Der Fehler ist eine Zahl, welche beschreibt, wie „schlecht“ ein künstliches neuronales Netz ein Problem löst.

4. Wie lernt ein künstliches neuronales Netz?

4.1. Grundlagen

Dieses Kapitel beschäftigt sich mit den Lernverfahren von künstlichen neuronalen Netzen. Grundsätzlich ist vorwegzunehmen, dass man unter „lernen“ lediglich die Optimierung der Parameter (der Gewichte und der Biases) versteht, um ein Problem bestmöglich lösen zu können, also mit minimalem Fehler. In folgendem Kapitel wird der wohl meist verwendete Lernalgorithmus erläutert: Das Gradientenabstiegsverfahren auf Basis von Backpropagation.

Vorweg wird das gesamte neuronale Netz inklusive Fehlerfunktion in einer Formel zusammengefasst. Das Ergebnis ist eine Formel, welche den Fehler des Netzes in Abhängigkeit zum Input zeigt.

$$f(W_1 \cdot \vec{a}) = \vec{a}_1 \quad f(W_2 \cdot \vec{a}_1) = \vec{a}_2 \quad \cdots \quad f(W_L \cdot \vec{a}_{L-1}) = \vec{a}_L \quad \sum_i \frac{1}{2}(a_i^L - t_i)^2 = E$$

$$E = ANN(\vec{a})$$

Abbildung 12: Zusammenfassung zu einer Formel, in Incskape und Word erstellt

Wobei W_1, W_2, …, W_L und $\vec{y}$ als Parameter dieser Funktion gesehen werden

Da aber für das Lernverfahren von Bedeutung ist, wie sich der Fehler in Abhängigkeit von den Gewichten verhält, wird das Ganze umgedreht. Die Gewichte werden zur unabhängigen Variablen und der Input wird ein Parameter der Funktion. Formelmäßig ändert sich hierbei nichts, es ist lediglich eine Sache der mathematischen Betrachtung, was als Variable und was als Parameter gesehen wird.

$$E = ANN(W_1, W_2, \dots, W_L)$$

In folgendem Diagramm ist diese Formel schemahaft illustriert. Es zeigt wie sich der Fehler E zu zwei Gewichten, also zwei Elemente aus den Gewichtsmatrizen verhält.

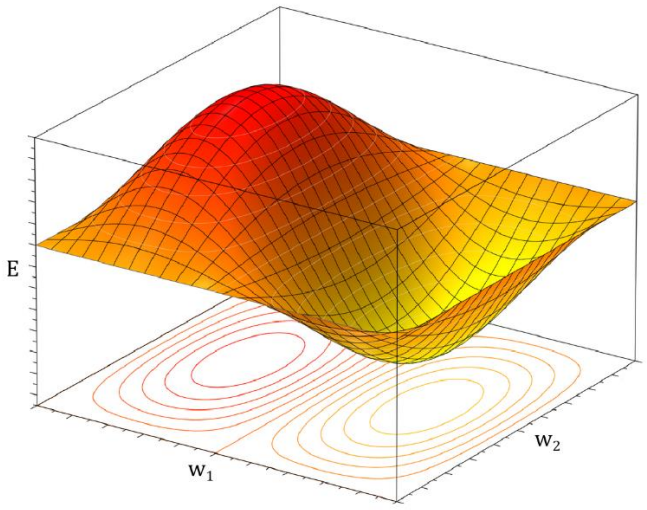

Abbildung 13: Fehler in Abbhängigkeit zu zwei Gewichten, Quelle: https://de.wikipedia.org/wiki/Datei:2D_Wavefunction_(2,1)_Surface_Plot.png, in Inkscape bearbeitet

Die Täler dieser Funktion sind Stellen, an denen der Fehler minimal ist. Also die entsprechenden Werte der Gewichte, an denen das Netz ein Problem optimal lösen kann. Beim Lernverfahren eines künstlichen neuronalen Netzes geht es darum, diese Täler zu finden. Bekanntermaßen sind die Täler einer zweidimensionalen Funktion einfach zu ermitteln. Doch gestaltet sich dies hier aufgrund der Vielzahl von Dimensionen, die ein neuronales Netz hat, deutlich schwerer. Beim Gradientenabstiegsverfahren nähert man sich dem Tal Schritt für Schritt (vgl. Rashid, 2017, S. 73). Man startet also an irgendeinen Punkt und ändert seine Position in kleinen Schritten talabwärts. Man kann sich das wie bei einem Ball vorstellen, der in einer hügeligen Landschaft bergab rollt.

Mathematisch gesehen werden die einzelnen Gewichte mit zufälligen Zahlenwerten besetzt. Daraufhin wird jeder dieser Werte so angepasst, dass sich die Gewichte Schritt für Schritt dem Minimum des Fehlers nähern. In folgender Formel ist dieser Sachverhalt für ein einzelnes Gewicht mathematisch dargestellt, wobei u die Richtung des Tals ist und l die Lernrate, eine Konstante, durch welche sich die Schrittweite einstellen lässt (vgl. Rashid, 2017, S. 73).

$$w_{neu} = w_{alt} + u * l$$

Die Ableitung einer Funktion ist für Abschnitte, in denen die Funktion steigt, positiv und für jene, in denen die Funktion fällt, negativ. So lässt sich u als die negative Ableitung des neuronalen Netzes and Stelle des Gewichtes und in Abhängigkeit zum Fehler schreiben. So besteht der finale Schritt darin, die Ableitung des neuronalen Netzes zu finden. Dafür wird der Backpropagation Algorithmus verwendet (vgl. Goodfellow u.a., 2018, S. 225).

4.2. Backpropagation

Als Backpropagation wird der Algorithmus zur Berechnung der Ableitung eines künstlichen neuronalen Netzes bezeichnet. Die Formeln hinter dem Algorithmus lassen sich durch Ableiten über Differenzierregeln herleiten.

Begonnen wird mit der Ableitung an Stelle eines Gewichts in der letzten Schicht: Im Folgenden sind die Formeln zur Berechnung der Ausgabe sowie des Fehlers dargestellt, wobei diese in drei Teile aufgeteilt ist. Des Weiteren werden hier keine Vektoren oder Matrizen verwendet, da diese das Ableiten erschweren. Achtung, hier ist die Benennung der Parameter leicht unterschiedlich: Die Ausgabe des Neurons wird als a_l bezeichnet. Wobei l für die Schicht steht, der das Neuron zugehörig ist und L steht für die letzte Schicht. Somit ist die Eingabe des Neurons a_{l-1}, da die Eingabe die Ausgabe der vorgeschalten Schicht ist.

$$E = \frac{1}{2}(a_L - t)^2 + \cdots$$

$$a_L = f(z)$$

$$z = a_{L-1}w + \cdots + b$$

Nun ist die Ableitung aufgrund der Kettenregel einfach zu bilden. Die Ableitung von jedem dieser Terme ist ein Faktor der Gesamtableitung. Das ist in der folgenden Formel dargestellt.

$$\frac{dE}{dw} = \frac{dE}{da_L}\frac{da_L}{dz}\frac{dz}{dw}$$

Der erste Teil ($\frac{dE}{da_L}$) ist durch die Potenzregel unkompliziert zu differenzieren. Hierbei sieht man, wieso das $\frac{1}{2}$ in der Kostenfunktion verwendet wird: Es hebt sich mit der zweiten Potenz in der Kostenfunktion auf, welche beim Ableiten heruntergeschrieben wird. Dadurch

bleibt ein konstanter Faktor erspart. Der folgende Term beschreibt die Ableitung an Stelle eines Neurons und wird auch als Teilfehler bezeichnet.

$$\frac{dE}{da} = \left(\frac{1}{2}(a_L - t)^2 + \cdots\right)' = (a - t)$$

Der zweite Teil ist lediglich die Ableitung der Aktivierungsfunktion. In dieser Arbeit wird die Sigmoidfunktion verwendet $f(x) = \sigma(x)$, da sie einfach zu differenzieren ist, im Gegensatz zur Stufenfunktion, die zum beispielsweise keine Ableitung hat. Für die Ableitung Sigmoid gilt: $\sigma'(z) = \sigma(z) * (1 - \sigma(z))$ und da in unserem Fall $\sigma(\mathrm{z}) = \mathrm{a}$ ist, kann man die Ableitung folgendermaßen beschreiben:

$$\frac{da}{dz} = \sigma'(z) = a * (1 - a)$$

Nun fehlt nur noch der letzte Teil:

$$\frac{dz}{dw} = (a_{L-1}w + \cdots + b)' = a_{l-1}$$

Wenn all diese Teile wieder zusammengefügt werden, erhält man die vollständige Ableitung.

$$\frac{dE}{dw} = (a - t) * a * (1 - a) * a_{l-1}$$

Hierbei darf nicht vergessen werden, dass es sich bei folgender Formel lediglich um das Differential eines Gewichts in der letzten Schicht handelt. Für ein Gewicht in einer anderen Schicht muss ein Ersatz für den Term $\frac{dE}{da}$ gefunden werden, da der Fehler nicht mehr direkt vom Output des Neurons abhängig ist. Ein Beispiel für ein Gewicht, welches sich nicht in letzter Schicht befindet, ist folgendem dreilagigen neuronalen Netz abgebildet. Achtung, hier wird die Schicht, in der sich das Neuron befindet, hochgestellt geschrieben, da die Neuronen einer Schicht tiefgestellt nummeriert werden.

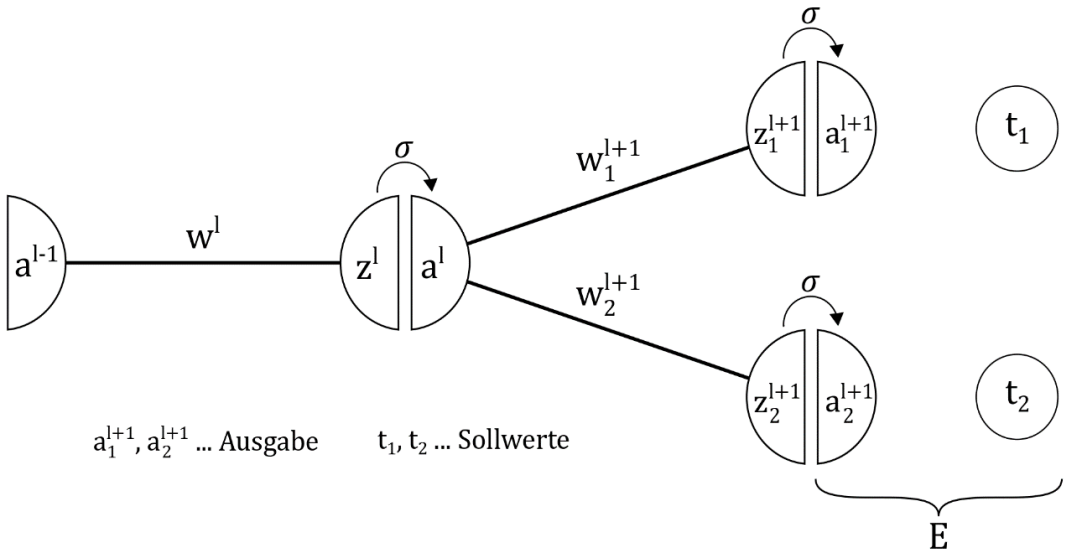

Abbildung 14: Gewicht in vorletzter Schicht, in Inkscape erstellt

Hier sind die Formeln zur Berechnung des Fehlers in Abhängigkeit des Neurons in vorletzter Schicht. Diese werden gebraucht, um die Ableitung des Netzes, an Stelle der Ausgabe des Neurons in vorletzter Schicht zu finden ($\frac{dE}{da^l}$). Diese Ableitung wird auch als Teilfehler bezeichnet.

$$E = \frac{1}{2}\left(a_1^{l+1} - t_1\right)^2 + \frac{1}{2}\left(a_2^{l+1} - t_2\right)^2$$

$$a_1^{l+1} = f\left(z_1^{l+1}\right) \qquad a_2^{l+1} = f\left(z_2^{l+1}\right)$$

$$z_1 = a^l w_1^{l+1} + b_1^{l+1} \qquad z_1 = a^l w_2^{l+1} + b_1^{l+1}$$

Daraus lässt sich folgende Formel für den gesuchten Teilfehler aufstellen:

$$\frac{dE}{da^l} = \frac{dE}{da_1^{l+1}}\frac{da_1^{l+1}}{da^l} + \frac{dE}{da_2^{l+1}}\frac{da_2^{l+1}}{da^l}$$

Verallgemeinert lässt sich dies folgendermaßen ausdrücken:

$$\frac{dE}{da^l} = \sum_k \frac{dE}{da_k^{l+1}}\frac{da_k^{l+1}}{da^l}$$

Ein zweiter Blick auf diese Formel lohnt sich. Hier steht wie man den Teilfehler eines beliebigen Neurons (mit Ausnahme jener in letzter Schicht) von den Teilfehlern der folgenden Schicht bildet. Das ist der letzte fehlende Puzzlestein zu einem lernenden neuronalen Netz. Im weiteren Teil wird der Teilfehler als ε^l und nicht mehr als Ableitung ($\frac{dE}{da^l}$) bezeichnet.

Als Erstes wird der Term $\frac{da_k^{l+1}}{da^l}$ im Summenausdruck nach bekannten Mustern aufgelöst.

$$\frac{da_k^{l+1}}{da^l} = \frac{da_k^{l+1}}{dz_k^{l+1}} \frac{dz_k^{l+1}}{da^l}$$

$$\frac{da_k^{l+1}}{dz_k^{l+1}} = \sigma'\left(z_k^{l+1}\right) = a_k^{l+1} * (1 - a_k^{l+1})$$

$$\frac{dz_k^{l+1}}{da^l} = \left(a^l w_{\mathrm{k}}^{l+1} + b_{\mathrm{k}}^{l+1}\right)' = w_k^{l+1}$$

$$\frac{da_k^{l+1}}{da^l} = a_k^{l+1} * \left(1 - a_k^{l+1}\right) * w_k^{l+1}$$

Nun können die Formeln für die Ableitung eines neuronalen Netzes erstmals vollständig formuliert werden.

$$\frac{dE}{dw^l} = \varepsilon^l * a^l * (1 - a^l) * a^{l-1}$$

Wenn sich das Neuron in letzter Schicht L befindet:

$$\varepsilon_k^L = a_k^L - y_k$$

Sonst:

$$\varepsilon^l = \sum_k \varepsilon_k^{l+1} * a_k^{l+1} * \left(1 - a_k^{l+1}\right) * w_k^{l+1}$$

Diese Formeln werden mithilfe von Vektoren und Matrizen nachgebildet. Weiterführende Informationen zum Thema Matrizen und Rechenregeln für Matrizen und Vektoren finden Sie im Anhang.

Begonnen wird mit der Formel zur Berechnung des Teilfehlers, der letzten Schicht.

$$\vec{\varepsilon}_L = \begin{pmatrix} a_1^L - y_1 \\ a_2^L - y_2 \\ a_3^L - y_3 \\ \vdots \end{pmatrix} = \vec{a}_L - \vec{y}_L$$

Die Formel zur Berechnung des Teilfehlers der anderen Schichten lässt sich durch die Verwendung der elementweisen Vektormultiplikation sowie dem Skalarprodukt folgendermaßen vereinfachen. $W_{:k}$ steht für alle Elemente der Gewichtmatrix in Spalte k und ist somit als Vektor zu sehen. Mit anderen Worten ist der Teilfehler eines Neurons nur von den Gewichten abhängig, die direkt die Ausgabe des Neurons beeinflussen.

$$\vec{\varepsilon}_l = \begin{pmatrix} W_{:1}^{l+1} \cdot [\vec{\varepsilon}_{l+1} \odot \vec{a}_{l+1} \odot (\begin{pmatrix} 1 \\ 1 \\ \vdots \end{pmatrix} - \vec{a}_{l+1})] \\ W_{:2}^{l+1} \cdot [\vec{\varepsilon}_{l+1} \odot \vec{a}_{l+1} \odot (\begin{pmatrix} 1 \\ 1 \\ \vdots \end{pmatrix} - \vec{a}_{l+1})] \\ \vdots \end{pmatrix}$$

Wird die Gewichtsmatrix transponiert, lässt sich diese Formel auch als Matrix-Vektor-Multiplikation schreiben.

$$\vec{\varepsilon}_l = W_{l+1}^T \cdot [\vec{\varepsilon}_{l+1} \odot \vec{a}_{l+1} \odot (\begin{pmatrix} 1 \\ 1 \\ \vdots \end{pmatrix} - \vec{a}_{l+1})]$$

Jedes Gewicht in einer Schicht hat eine eigene Kombination von einer Eingabe, die es verstärkt beziehungsweise schwächt, sowie einem Neuron, dem es zugehörig ist. Da bei der Formel zur Aktualisierung der Gewichte a_k^{l-1} für die Eingabe steht und $\varepsilon^l * a^l * (1 - a^l)$ für das zugehörige Neuron, wird jedes Gewicht in der Gewichtsmatrix über eine eigene Kombination von diesen Werten aktualisiert. Diese Matrix zum Aktualisieren kann somit durch die Verwendung des dyadischen Produkts berechnet werden.

$$\frac{dE}{dW_l} = \begin{pmatrix} \varepsilon_1^l * a_1^l * \left(1 - a_1^l\right) * a_1^{l-1} & \varepsilon_1^l * a_1^l * \left(1 - a_1^l\right) * a_2^{l-1} & \cdots \\ \varepsilon_2^l * a_2^l * \left(1 - a_2^l\right) * a_1^{l-1} & \varepsilon_2^l * a_2^l * \left(1 - a_2^l\right) * a_2^{l-1} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

$$= \vec{\varepsilon}_l \odot \vec{a}_l \odot (\begin{pmatrix} 1 \\ 1 \\ \vdots \end{pmatrix} - \vec{a}_l) \otimes \vec{a}_{l-1}$$

Diese Formeln lassen sich noch etwas effizienter gestalten. Beispielsweise wird der Teilfehler dazu verwendet, die Aktualisierungen der zugehörigen Schicht, sowie den Teilfehler der vorgeschalteten Schicht, zu berechnen. In beiden Fällen wird der Teilfehler nochmals mit der Ableitung der Sigmoidfunktion multipliziert. Multipliziert man diese gleich zum Teilfehler, erspart man sich einen Rechenschritt pro Schicht. Daraus ergeben sich folgende aktualisierten Formeln:

$$\frac{dE}{dW_l} = \vec{\varepsilon}_l \otimes \vec{a}_{l-1}$$

$$\vec{\varepsilon}_L = (\vec{a}_L - \vec{y}_L) \odot \vec{a}_L \odot (\begin{pmatrix} 1 \\ 1 \\ \vdots \end{pmatrix} - \vec{a}_L)$$

$$\vec{\varepsilon}_l = W_{l+1}^{\mathrm{T}} \cdot \vec{\varepsilon}_{l+1} \odot \vec{a}_l \odot (\begin{pmatrix} 1 \\ 1 \\ \vdots \end{pmatrix} - \vec{a}_l)$$

Im Wesentlichen beschreiben diese drei Formeln die Ableitung des neuronalen Netzes an Stelle der Gewichte in Abhängigkeit zum Fehler des Netzes. Sie stellen den Backpropagation- Algorithmus da, den Kern des Lernverhalten eines neuronalen Netzes. Wie sieht der Einsatz dieser Formeln in der Praxis aus? Allererst rechnet der Roboter ein Trainingsbeispiel, ganz normal als Input und speichert alle Ausgabewerte jeder Schicht zwischen. Ist dies fertig gerechnet, startet der Roboter den Backpropagation Algorithmus: Anfangs berechnet er den Teilfehler der letzten Schicht. Dieser wird für die Aktualisierung der Gewichte in der letzten Schicht verwendet, aber auch um den Teilfehler der vorgeschalteten Schicht zu berechnen. Der hierbei berechnete Teilfehler wird wiederum verwendet, um die Gewichte der vorletzten Schicht anzupassen sowie den Teilfehler der vorgeschalteten Schicht zu berechnen. Deshalb wird dieser Algorithmus auf Deutsch auch oft *Fehlerrückführungsmethode* genannt (vgl. Kinnebrock, 1992, S. 41).

Kernideen:

- Beim Lernverfahren eines künstlichen neuronalen Netzes geht es darum, den Fehler durch Anpassen der Gewichte zu minimieren. Also mit anderen Worten geht es um das Finden des Minimums der Fehlerfunktion.
- Dafür wird das Gradientenabstiegsverfahren verwendet. Hier nähert man sich dem Tal Schritt für Schritt.
- Die Richtung des Tales gibt die negative Ableitung an. Um diese zu ermitteln wird der Backpropagation Algorithmus verwendet.

5. Simulation am Computer

Das folgende Kapitel widmet sich der Implementierung eines neuronalen Netzes in ein Computersystem. Dieses Netz wird im Folgenden darauf trainiert, handgeschriebene Ziffern richtig zu klassifizieren.

5.1. Wieso Python?

Um das neuronale Netz auf dem Computer umsetzen zu können, wird in dieser Arbeit die Programmiersprache Python verwendet. Python hat den Vorteil, dass es einfach erweiterbar ist. So gibt es eine Vielzahl von Modulen, welche zum Beispiel das Rechnen mit Matrizen oder das Erstellen von Graphen erheblich vereinfachen. Es hat auch eine einfache sowie übersichtliche Syntax und ist kostenfrei.

5.2. Wieso die Erkennung handgeschriebener Zahlen?

Die Aufgabe des Netzes ist es, handgeschriebene Ziffern zu erkennen. Das Problem ist unscharf und komplex genug, dass es ein Computer nicht auf einem herkömmlichen Rechenweg lösen kann. Hierfür gibt es eine Datenbank, die oft von Forschern im Bereich der künstlichen Intelligenz verwendet wird, um ihre neusten Ideen auszuprobieren und zu vergleichen. Es gibt eine Liste, auf der die Ergebnisse von Experten eingetragen sind. So ist es immer möglich, die Ergebnisse des in dieser Arbeit erstellten Netzes mit denen von Experten zu vergleichen. Die Datenbank heißt MNIST und beinhaltet einen Trainingsdatensatz von 60.000 Bildern handgeschriebener Ziffern, sowie einen Testdatensatz mit 10.000 Bildern. All diese sind für Trainingszwecke bzw. Überprüfungszwecke bereits mit der korrespondierenden Ziffer beschriftet (vgl. LeCun u.a., 2013). Im Folgenden sind drei Bilder aus der Datenbank dargestellt.

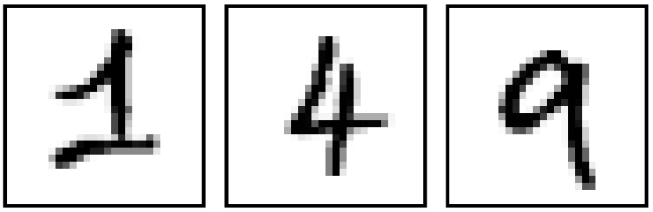

Abbildung 15: Beispielziffern der MNIST Datenbank, Quelle: http://yann.lecun.com/exdb/mnist/, in Python visualisiert

5.3. Das Programm

5.3.1. Wichtige Begriffe

Bevor das Programm selbst erklärt werden kann, müssen ein paar zentrale Begriffe klargelegt werden.

Variablen stehen in der Informatik für kleine Speicherplätze, in die man Objekte wie unter anderem eine Zahl, ein Text oder einen Vektor speichern kann.

Funktionen sind in der Informatik ähnlich wie in der Mathematik: Für Eingaben wird eine Ausgabe erzeugt. Nur, dass in der Informatik Funktionen nicht nur über mathematische Ausdrücke definiert werden können, sondern auch über jegliche andere Programmabschnitte. Diese werden beim Aufrufen der Funktion ausgeführt.

Eine **Klasse** ist der „Bauplan“ eines Objektes. So lassen sich zum Beispiel aus der Klasse „Hund“ unterschiedliche Hunde (Objekte) ableiten. Die Hunde haben zwar unterschiedliche Eigenschaften, aber die gleichen grundlegenden Fähigkeiten (bellen, laufen, etc.). In der Informatik werden die Eigenschaften durch Variablen beschrieben und heißen „Attribute“. Die Fähigkeiten werden durch Funktionen dargestellt und heißen Methoden.

5.3.2. Das Programm und ich

Ich bin bei dem Erstellen des Programms anfangs unstrukturiert vorgegangen. Ich begann mit einer Hauptklasse, welcher ich nach und nach um die Attribute und Methoden eines künstlichen neuronalen Netzes erweiterte. Durch den unstrukturierten Programmierprozess und stundenlangen Fehlersuchen ist ein großes Chaos entstanden, so dass ich entschied das Ganze in einer neuen Datei noch einmal schön und strukturiert zu schreiben. In diesem Schritt wurde herausgestrichen, zusammengefasst, umbenannt und aufgespalten. Der Informatiker würde dazu sagen: „Das Programm wurde aufgeräumt.“ Zum Schluss wurde im Programm selbst, das Programm beschrieben (es wurde kommentiert), dass nicht nur ich selbst wieder verstehe was ich da geschrieben habe, sondern auch jeder andere Informatikaffine Mensch soll das Programm verwenden und begreifen können.

5.3.3. Struktur und Aufbau

Wie beschrieben besteht das Programm hauptsächlich aus einer Klasse. Diese heißt *ANN*, das ist kurz für „artifical neural network“. Im Folgenden sind die Attribute und Methoden dieser Klasse aufgelistet.

- Attribute:
    - *weights*: Eine Liste aller Gewichtsmatrizen des Netzes
- Methoden:
    - *__init__*: Wird beim Ableiten eines Objektes aufgerufen, hier werden die Gewichstmatrizen deklariert.
    - *run*: Lässt das Netz für einen Eingabewert durchlaufen
    - *learn*: Algorithmus für das Lernverfahren des Netzes
    - *eval_accuracy*: Berechnet die Genauigkeit des Netzes.
    - *feed_backward*: Rückwärtsabfrage (wird später erklärt)
    - *sigmoid*: Sigmoidfunktion
    - *backprob*: Backpropagation Algorithmus

Die ganzen Funktionen arbeiten hauptsächlich mit Vektoren und Matrizen als Variablentypen. Dies erlaubt ein sehr reduziertes Programm sowie die unveränderte Verwendung der hergeleiteten Formeln. Da Python von selbst keine Vektoren- beziehungsweise Matrizenrechnung unterstützt, wird das Modul Numerical Python verwendet.

Zudem gibt es im Programm, neben der Klasse *KNN*, noch zwei weitere Funktionen. Einerseits *format_mnist* und andererseits *main*. format_mnist wird verwendet, um die MNIST Datensätze zu importieren und in das erforderliche Format zu bringen. Da aber Klassen und Funktionen irgendwo aufgerufen werden müssen, gibt es die Funktion *main*. Diese startet die beschriebene Klasse und Funktion unter deklarierten Parametern. Die Funktion *main* startet von allein, wenn das Programm aufgerufen wird.

5.4. Optimierung der Parameter

Der Lernalgorithmus verwendet verschiedene Parameter, welche im Vorhinein zu deklarieren sind. Dieses Unterkapitel beschäftigt sich mit dem Finden der optimalen Werte für diese Parameter.

5.4.1. Lernrate

In folgendem Diagramm ist die Trefferquote des selbstprogrammierten Netzes für unterschiedliche Lernraten abgebildet. Man kann erkennen, dass zu kleine oder zu große

Lernraten zu schlechten Ergebnissen führen. Das ist auch logisch, da man durch kleine Lernraten zu kleine Schritte macht und damit die Geschwindigkeit des Gradientenabstieges begrenzt. Auf der anderen Seite führen zu große Lernraten zu einem Pendeln um das Minimum, da die Schritte immer über das Minimum hinausschießen. Die optimale Trefferquote für das beschriebene Problem erreicht eine Lernrate von 0.3.

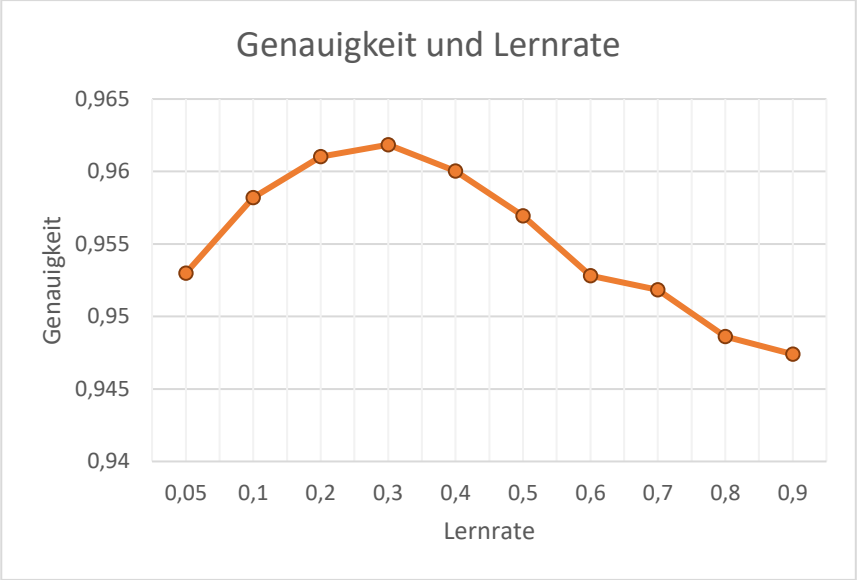

Abbildung 16: Genauigkeit und Lernrate, in Word erstellt

5.4.2. Epochen

Eine bessere Genauigkeit kann erzielt werden, wenn der Lernprozess mehrmals wiederholt wird. Die Anzahl dieser Wiederholungen nennt man Epochen. In folgenden Diagrammen sieht man, dass die Trefferquote anfangs beachtlich steigt, aber dann konstant bleibt. Der bleibende Fehler kommt von Ungenauigkeiten wie ein Pendeln um das Minimum, uneindeutigen Testbildern oder durch schlicht eine zu kleine Anzahl an Trainingsbeispielen.

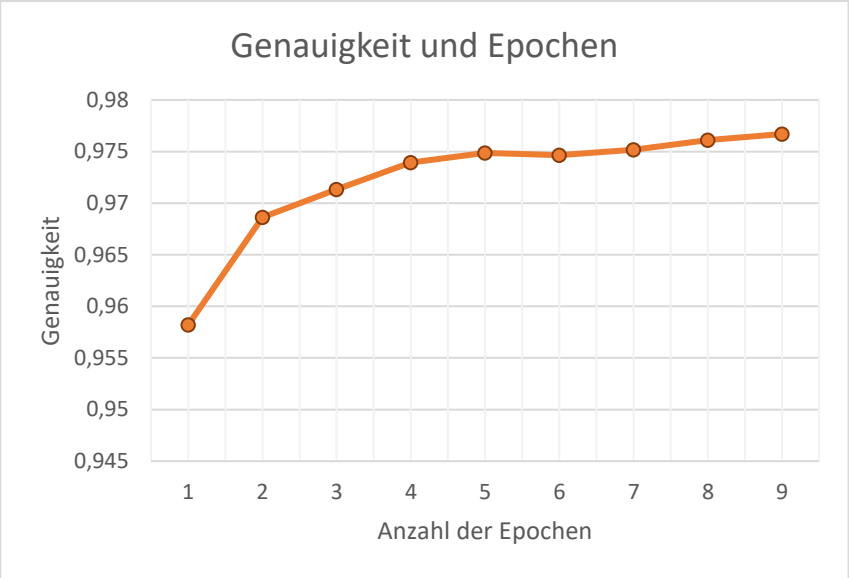

Abbildung 17: Genauigkeit und Epochen, in Word erstellt

5.5. Rückwärtsabfrage

Was passiert eigentlich in einem neuronalen Netz? Es lernt, selbstständig große Datensätze richtig zu klassifizieren, doch die selbstgebildeten Regeln, nach denen das Netz vorgeht, sind von außen nicht einsehbar. Das ist eigentlich kein Problem, wenn man lediglich an Antworten interessiert ist, doch ist es beispielsweise für Überprüfungszwecke vorteilhaft, einen Einblick in das Gedächtnis eines neuronalen Netzes zu bekommen.

Hier kommt die Rückwärtsabfrage ins Spiel. Die Idee ist, dass man, anstatt eine Eingabe von vorne nach hinten durchzugeben, eine gewünschte Ausgabe von hinten nach vorne durch das Netz schleust (vgl. Rashid, 2017, S. 170). Die Ausgabe kann zum Beispiel für die Ziffer Null stehen. Leitet man diese von hinten nach vorne durch das Netz, erhält man folgendes Bild.

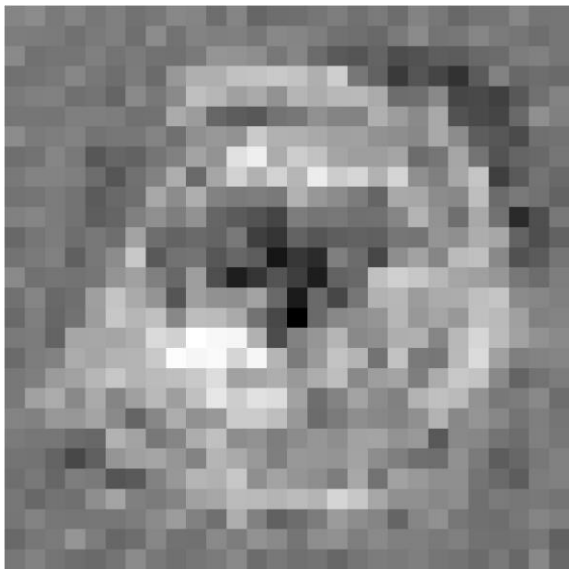

Abbildung 18: Rückwärtsabfrage der Ziffer Null, in Python visualisiert

Dieses Bild ist das Bild, welches das Netz bestmöglich als Null klassifizieren kann. Also sozusagen für das Netz das Idealbild einer Null. Daraus lässt sich so einiges ablesen:

- Deckt die Spur des Stiftes am Eingabebild, die hellen Stellen in diesem Bild ab, so kann das Netz das Bild bestmöglich als Null klassifizieren. Das ist ja auch ganz logisch, da die hellen Stellen im Kreis einer Null verlaufen.
- Die dunklen Stellen müssen an einem Eingabebild freibleiben, um bestmöglich als Null erkannt zu werden. Diese befinden sich hauptsächlich in der Mitte des Ringes, wo bei einer Null natürlich nichts ist.
- Das Netz steht den grauen stellen gleichgültig gegenüber. Diese beeinflussen die Ausgabe kaum.

Im Folgenden sind auch die Bilder aller anderen Ziffern ebenfalls über die Rückwärtsabfrage konstruiert.

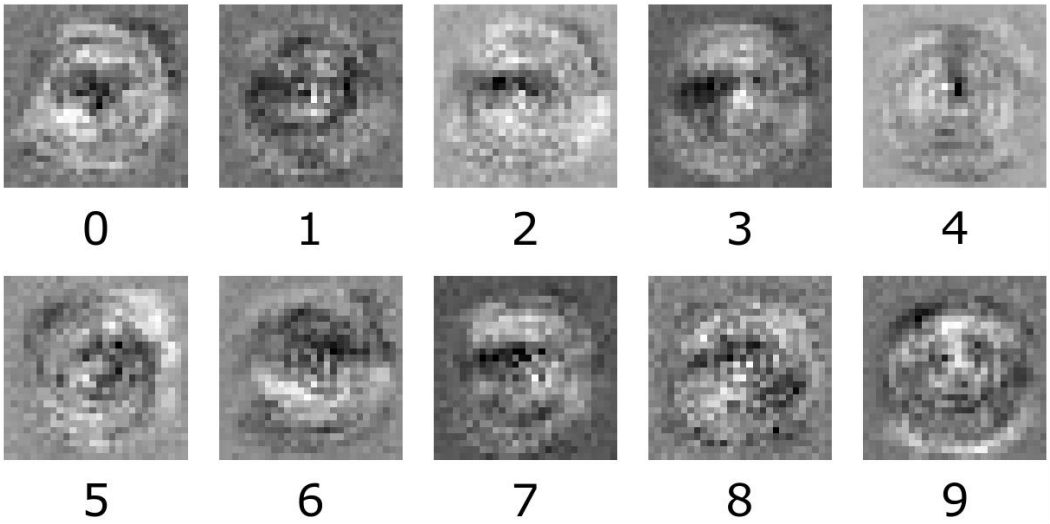

Abbildung 19: Rückwärtsabfrage aller Ziffern, in Python visualisiert

Auf den Bildern mancher Ziffern lassen sich die ursprünglichen Ziffern über helle Bereiche noch erkennen. Während sich das Netz bei den anderen mehr auf die Bereiche konzentriert hat, die frei bleiben müssen. Das ist auch in Ordnung, schließlich funktioniert es und das Netz erkennt die Ziffern mit einer Genauigkeit von über 95%

5.6. Datenvervielfältigung

Die zuvor generierten Bilder sind das Gedächtnis des neuronalen Netzes. Sie sind aus dem Lernen von 60 000 Trainingsdaten entstandenen. Erst eine große Anzahl an Daten ermöglicht dem Netz, Bilder mit solch einer Genauigkeit zu erkennen (vgl. Rashid, 2017, S. 174). Will man die Genauigkeit erhöhen, gibt es vielerlei Tricks, die einerseits eine interne Ungenauigkeit des Netzes ausgleichen, wie die Anpassung der Lernrate, oder die Trainingsdaten besser ausnützen, wie das Lernen in Epochen. Doch meist hilft kaum etwas mehr als das Lernen mit mehr Trainingsdaten.

Doch woher bekommt man diese Daten? Selbst Zehntausende von Ziffern zu schreiben und zu scannen ist sehr mühsam und zeitaufwendig. Da wäre es doch praktisch, wenn es einen Weg gäbe, aus den bereits vorhandenen Daten neue zu generieren. Den gibt es. Ein Ansatz dafür ist einfach das Drehen der Trainingsdaten um einen kleinen Winkel nach vor sowie zurück (vgl. Rashid, 2017, S. 174). Dadurch bleibt die Zahl im Wesentlichen gleich, jedoch ändern sich die Pixeldaten für den Computer.

Die Bilder werden 10° nach links sowie 10° nach rechts gedreht. Dadurch hat man insgesamt die dreifache Menge an Trainingsdaten, und dank der vielen Erweiterungen

von Python ist das sehr einfach und schnell getan. Als Beispiel ist hier das Bild einer Eins inklusive der gedrehten Variationen.

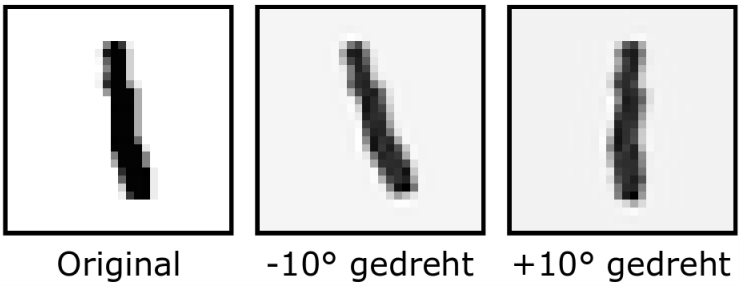

Abbildung 20: Eine Eins der MNIST Datenbank inklusive gedrehter Variationen, Quelle: http://yann.lecun.com/exdb/mnist/, in Python bearbeitet

Das Ergebnis kann sich sehen lassen. Das trainierte Netz erreicht eine Genauigkeit von 98,89% bei 30 Epochen. Das Netz hatte drei Schichten: Zwei mit 500 Neuronen und eine Ausgabeschicht mit 10 Neuronen. Es kann mit den Ergebnissen von Experten mithalten, dessen dreilagige neuronale Netze auf der Liste der MNIST Datenbank auf ähnliche Genauigkeiten kommen (vgl. LeCun u.a., 2013).

Kernideen:

- Das Programm besteht aus einer Hauptklasse, welche um die Attribute und Methoden eines künstlichen neuronalen Netzes erweitert wurde.
- Um einen kleinen Einblick in das Gedächtnis eines neuronalen Netzes zu bekommen, lässt sich über die Rückwärtsabfrage berechnen, welche Eingaben das Netz für ein bestimmtes Ergebnis negativ bzw. positiv beeinflussen.
- Durch Rotieren der Bilder lassen sich die Trainingsdaten vervielfachen, was zu einer besseren Genauigkeit des Netzes führt.

6. Ethische und gesellschaftliche Probleme künstlicher neuronaler Netze

Künstliche neuronale Netze werden in vielen Bereichen eingesetzt, welche eine direkte Auswirkung auf Menschen haben. Zum Beispiel entscheiden sie, ob Kredite vergeben werden und beurteilen die Angeklagte beziehungsweise den Angeklagten vor Gericht (vgl. Krumay, 2018, S. 142). Auch wenn sie offiziell meist nur als „unverbindliche Unterstützung“ eingesetzt werden, vertrauen viele Menschen diesen Maschinen (vgl. Spiekermann, 2016). Das Problem ist, dass die eingesetzten Systeme meist eine „Blackbox“ sind (vgl. Castelvecchi, 2016). So weiß der Anwender und auch oft der Programmierer nicht, aus welchen Kriterien das Netz seine Schlüsse gezogen hat. So ist es oftmals schwer herauszufinden, ob das Netz falsche Schlüsse zieht oder bestimmte Personengruppen bevorzugt. Des Weiteren ist es kaum nachvollziehbar, ob und wie sich ein künstliches neuronales Netz über bestimmte Eingaben gezielt manipulieren lässt (vgl. Castelvecchi, 2016). Wie sich künstliche neuronale Netze irren können, ist in folgendem Unterkapitel angeschnitten.

6.1. Fehlschlüsse künstlicher neuronaler Netzwerke

6.1.1. Voreingenommene Daten

Wissenschaftler der Universität von Washington wollten herausfinden, wieso ihr künstliches neuronales Netz ein Bild eines Huskys als Wolf erkennt. Sie konnten rekonstruieren, nach welchen Teilen des Bildes das System geschlossen hat, als es das Bild klassifizierte, und kamen zum Schluss, dass das Netz ausschließlich nach dem Schnee im Hintergrund des Bildes geurteilt hat. Das Problem war, dass fast alle Bilder von Wölfen, nach denen das Netz gelernt hat, in einer verschneiten Umgebung aufgenommen wurden. So hat das Netz von Schnee im Hintergrund auf einen Wolf geschlossen. (vgl. Ribeiro u.a., 2016) Ein ähnliches Problem gibt es in den Gesichtserkennungssystemen von Microsoft und IBM. Diese erkennen die Gesichter von Männern mit niedrigerer Fehlerquote als die von Frauen, und Personen mit heller Hautfarbe werden am besten erkannt (vgl. Tilly, 2018, S. 139). Dies kommt höchstwahrscheinlich daher, dass mit einem Datensatz gelernt wurde, in dem viele männliche Gesichter mit heller Haut waren. Ein Problem wird dies aber dann, wenn neuronale Netze über eine Kreditvergabe entscheiden oder Angeklagte in Gerichtsprozessen beurteilen. Zum Beispiel wird in amerikanischen Gerichten die Software

COMPAS verwendet. Diese beurteilt wie wahrscheinlich es ist, ob Verurteilte wieder eine Straftat begehen. Eine Studie fand heraus, dass diese doppelt so wahrscheinlich fälschlicherweise Menschen mit dunkler Hautfarbe als risikoreicher einstuft (vgl. Angwin u.a., 2016). Solch Fehlverhalten wurde nicht absichtlich von böswilligen Programmierern provoziert. Das Problem ist, dass künstliche neuronale Netze zum Lernen sehr viele Daten brauchen So ist es beinahe unmöglich, einen Überblick über ihre Qualität zu behalten.

6.1.2. Ungewohnte Daten

Fast alle Modelle neuronaler Netze sind sehr instabil gegenüber ungewohnten Mustern oder Rauschen (vgl. Tilly, 2018, S. 139). Wird zum Beispiel über ein Bild ein bestimmtes Rauschen gelegt, so wird es komplett falsch klassifiziert, obwohl ein Mensch beinahe keinen Unterschied merken würde. Des Weiteren versuchen selbst hochentwickelte Systeme in eine Eingabe eine Antwort zu interpretieren. So wird eine schwarz/grau gemusterte Fläche mit hoher Sicherheit als Königspinguin klassifiziert und eine rot/weiß gestreifte als elektrische Gitarre (vgl. Nguyen u.a., 2015). Das mag vielleicht nach harmlosen Verwechslungen klingen, doch lassen sich diese Schwachstellen auch manipulativ für gezielte Missinterpretationen einsetzen. Ein Beispiel hierfür wäre, dass Computerwissenschaftler ein unhörbares Rauschen über eine Audiobotschaft gelegt haben, wodurch Spracherkennungssysteme diese Botschaft ganz anders auffassen als Menschen (vgl. Cisse u.a., 2017). So ist es möglich, bei jemanden, der einen Sprachassistenten besitzt, anzurufen und eine Botschaft auf den Anrufbeantworter zu sprechen. Für den Menschen möge sich diese wie ein Gruß anhören, aber sie ist so manipuliert, dass der Sprachassistent diese als Bestellung interpretiert.

6.2. Artificial Ethics

6.2.1. Grundlagen

Selbstlernende Maschinen handeln immer mehr selbstständig und unabhängig von Menschen. Vor allem bei der autonomen Fahrzeugsteuerung kann es zu Situationen kommen, in denen ein Auto Entscheidungen um Leben und Tod treffen muss. Man hofft dabei auf die Entwicklung eines allgemeinen Moralkodexes für Maschinen (vgl. Krumay, 2018, S. 142). In diesem Zusammenhang werden oftmals die Asimovschen Gesetze zitiert. Diese wurden von einem Romanautor erstellt und sind selbstverständlich nicht für die Grundlage einer Roboterethik gedacht. Dennoch werden sie oft als dessen Basis gesehen.

0. *Ein Roboter darf der Menschheit keinen Schaden zufügen oder durch Untätigkeit zulassen, dass der Menschheit Schaden zugefügt wird.*

1. *Ein Roboter darf einem menschlichen Wesen keinen Schaden zufügen oder durch Untätigkeit zulassen, dass einem menschlichen Wesen Schaden zugefügt wird.*

2. *Ein Roboter muss Befehlen gehorchen, die ihm von Menschen erteilt werden, es sei denn, dies würde gegen das erste Gesetz verstoßen.*

3. *Ein Roboter muss seine eigene Existenz schützen, solange solch ein Schutz nicht gegen das erste oder zweite Gebot verstößt.* (Scholtysek, 2015)

Diese Regeln sind jedoch viel zu unzureichend und eindimensional, um die menschliche Moral abzubilden. Im Folgenden wird ein weit verbreitetes Beispiel präsentiert, welches komplexere Regeln verlangt.

6.2.2. Das Trolley Problem

Schon in den 1930er Jahren skizzierte Philippa Foot das „Trolley Problem". Es ist ein Gedankenexperiment, bei dem eine unhaltbare Straßenbahn auf eine Gruppe von Menschen zurast, jedoch kann der Weichensteller die Weiche umlegen, sodass die Straßenbahn der Menschenmenge ausweicht. Doch auf der Ausweichstrecke befindet sich auch eine Person, die in diesem Fall sterben würde. Dieses Problem kann als „moralisches Dilemma" bezeichnet werden, da in beiden Fällen, die Weiche wird umgelegt oder nicht, eine Regel aus dem Moralkodex gebrochen wird. Einerseits lässt man die Menschen durch Nichtstun sterben und andererseits bringt man den anderen Menschen aktiv um. Dieses Gedankenexperiment lässt sich auch auf intelligente Maschinen neu definieren im Kontext des automatisierten Fahrens. Das Grundproblem ist folgendes: Ein selbstfahrendes Auto rast unhaltbar auf eine Betonbarriere zu. Im Falle des Nichtstuns würde der Fahrzeuginsasse ums Leben kommen. Es kann ihr aber auch ausweichen, jedoch würde das Fahrzeug dabei Menschen überfahren, welche die Straße auf einem Zebrastreifen überqueren. Das MIT untersucht dieses Problem, in dem sie die Daten von Menschen sammelt, welche auf der Website der sogenannten „Moral Machine" selbst urteilen können (vgl. Krumay, 2018, S. 144).

Es gibt verschiedene Lösungsansätze zum Trolley Problem von selbstfahrenden Fahrzeugen, welche sich ebenfalls über Regeln in eine Maschine implementieren lassen können.

Ein weit umstrittener Ansatz ist, immer den Fahrzeuginsassen zu schützen. Diese Forderung steht im Gegensatz zu dem generellen Schutz der Unschuldigen. Weitere Diskussionen gibt es über eine Entscheidung je nach Wert der Personen für die Gesellschaft. (vgl. Krumay, 2018, S. 145)

6.2.3. Probleme der Deklaration

Hier stellt sich die Frage, ob sich unsere Moral eindeutig in Regeln ableiten lässt und ob es damit einen Moralkodex für Maschinen überhaupt gibt, denn dieser muss auf einem allgemein anerkannten Moralkodex basieren (vgl. Krumay, 2018, S. 145). Wir Menschen leiten unsere Moral im Laufe unseres Lebens von der umgebenden Gesellschaft ab. Doch wäre dies für Maschinen nicht denkbar, da man kaum einen Einfluss auf die Qualität der Quellen, aus denen die Maschine lernt, hat. Des Weiteren kann man den Lernprozess einer Maschine nicht mit dem eines Kindes vergleichen. Einerseits lernen Maschinen unreflektiert, was zu lernen vorgesehen ist, und andererseits geht von ihnen eine ganz andere Gefahr aus. So stellen Sie sich eine Drohne oder ein Auto vor, das erst lernen muss, was richtig oder falsch ist.

Hier muss meines Erachtens noch etwas eingeschoben werden. In diesem Unterkapitel wurde über einen Moralkodex für Maschinen gesprochen. Dieser darf aber nicht mit einem „Moralempfinden“ verwechselt werden. Aktuelle selbstlernende Systeme sind nicht intelligent, sie handeln lediglich auf Basis von Daten, durch welche sie optimiert wurden. Ein Moralkodex kann entweder ein Datensatz von Situationen sein, aus denen das System lernt, oder klar formulierte Regeln, nach denen es handelt.

6.3. Responsible Research and Innovation

Künstliche neuronale Netze (und im weiteren Sinn „künstliche Intelligenz“) ist ein Bereich mit viel Zukunft und Potenzial. Um die Forschung verantwortungsvoll zu betreiben, hat die Europäische Union einen Standard veröffentlicht, nach dem sich Universitäten in ganz Europa richten. Das Ziel ist es Forschung frei zugänglich zu machen und Fehlentwicklungen vorzubeugen. Das Modell heißt Responsible *Research and Innovation* oder kurz RRI. (vgl. Krumay, 2018, S. 145) Der Ansatz gliedert sich in vier Prozessdimensionen:

1. *Vielfältig und integrierend*
2. *Antizipativ und reflektiv*

3. *Offen und transparent*

4. *Reaktionsfähig und adaptiv bei Veränderungen* (Krumay, 2018, S. 145)

In der ersten Prozessdimension geht es darum möglichst viele Steakholder in den Forschungsprozess einzubinden. Diese bringen nicht nur unterschiedliche Bedürfnisse, sondern auch unterschiedliches Wissen mit. Dadurch soll ein gemeinsamer Entwicklungsprozess entstehen. (vgl. ebd., S. 145)

Die zweite Prozessdimension soll gesellschaftliche Katastrophen, wie beispielsweise durch Nobel oder Oppenheimer verursacht, verhindern. In dieser Dimension geht es um ein selbstkritisches Hinterfragen der Innovation, sowie um ein Reflektieren und Abwägen über mögliche Auswirkungen. (vgl. ebd., S. 145)

Über die dritte Prozessdimension soll der Gesellschaft die Forschung zugänglich gemacht werden und ihnen somit die Verunsicherung gegenüber den neuen Entwicklungen nehmen. (vgl. ebd., S. 146)

Die letzte Prozessdimension verlangt die Bereitschaft von Personen und Organisationen, sich veränderten Situationen anpassen zu können. (vgl. ebd., S. 146)

6.4. Zusammenfassung

Künstliche neuronale Netzwerke stellen eine beispiellose Entwicklung dar und werfen somit einige ethische und gesellschaftliche Probleme und Fragen auf, welche in dieser Praxisnähe noch nie gestellt wurden. Ebenso haben auch diese neuen Techniken ihre eigenen Gefahren und Risiken, denen man sich unvoreingenommen stellen muss, ohne dabei an eine wirkliche Intelligenz zu glauben. Auch ist es wichtig, in einem Gebiet mit so viel Potential die Forschung gewissenhaft und rücksichtsvoll voranzubringen, um nicht zuletzt negativen Auswirkungen auf die Gesellschaft vorzubeugen.

7. Resümee

Künstliche neuronale Netze setzen sich aus parallel und seriell geschalteten künstlichen Neuronen zusammen. Diese sind ein mathematisches Modell biologischer Nervenzellen. Vergleichbar mit biologischen Reizen, werden im Modell Zahlenwerte von hinten nach vorne durch das Netz gereicht. Synapsen und Schwellenwerte, modelliert als Gewichte und Biases, bestimmen, wie diese Reize verarbeitet werden.

Über eine sogenannte „Fehlerfunktion“ kann der Fehler eines künstlichen neuronalen Netzes ermittelt werden, dieser beschreibt wie inkorrekt das Netzwerk eine Aufgabe löst. Das Lernverfahren sucht die Gewichts- und Biaswerte am Minimum der Fehlerfunktion. In dieser Arbeit wurde das Gradientenabstiegsverfahren als Lernalgorithmus beschrieben. Über dieses werden die Parameter Schritt für Schritt Richtung Minimum des Fehlers optimiert. Die Richtung des Tales gibt die negative Ableitung an.

Die beschriebe Theorie wurde in einem selbstgeschriebenen Programm angewandt. Das Ziel, handgeschriebene Ziffern zu klassifizieren, konnte mit einer beachtlichen Genauigkeit von 98,89% umgesetzt werden. Das war nicht zuletzt dem Vervielfachen der Trainingsdaten zu verdanken. Diese Daten, welche aus Bildern handgeschriebener Ziffern bestehen, wurden durch das Erstellen leicht gedrehter Variationen verdreifacht.

Das Hauptproblem künstlicher neuronaler Netzwerke liegt in deren Intransparenz. So ist es für den Benutzer eines selbstlernenden Systems meist nicht einsehbar, worauf basierend die Maschine ihre Entscheidungen trifft. Maschinen übernehmen aufgrund maschinellen Lernens auch immer mehr verantwortungsvolle Aufgaben, welche bis jetzt ausschließlich Menschen vorbehalten waren. Die Frage, ob Maschinen ethisch handeln können, bleibt jedoch bestehen.

Das in dieser Arbeit behandelte Modell eines neuronalen Netzes ist die Basis für zahlreiche Weiterentwicklungen und Optimierungen. Sie stellen einen bedeutenden Bestandteil des Teilgebietes „künstlicher Intelligenz“ in der Informatik da. Diese Arbeit zeigt, dass künstliche Intelligenz wohl doch noch mehr „künstlich“ als „intelligent“ ist. Und doch haben künstliche neuronale Netze ihre eigenen Gefahren, denen man sich unvoreingenommen stellen muss, ohne dabei von weit futuristischeren Problemen, welche die Filmindustrie aufwirft, abgelenkt zu sein.

Literaturverzeichnis

Alpaydin, Ethem: Maschinelles Lernen. 2. Auflage. Berlin/Boston: *Walter de Gruyter GmbH*, 2019.

Angwin, Julia u.a.: Machine Bias. 23.5.2016. https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing [Zugriff. 2.2.2020].

Beutelsbacher, Albrecht: Lineare Algebra. Eine Einführung in die Wissenschaft der Vektoren, Abbildungen und Matrizen. 7. Auflage. Wiesbaden: *Vieweg+Teubner | GWV Fachverlage GMBH*, 2010.

Castellvecci, Davide: Eine tückische Blackbox. 16.11.2016. https://www.spektrum.de/news/eine-tueckische-blackbox/1429906 [Zugriff: 2.2.2020].

Cisse, Moustapha u.a.: Houdini: Fooling Deep Structured Prediction Models. 17.7.2017. https://arxiv.org/abs/1707.05373 [Zugriff: 2.2.2020].

Ertel Wolfgang: Grundkurs Künstliche Intelligenz. Eine praxisorientierte Einführung. Wiesbaden: *Vieweg & Sohn Verlag | GWV Fachverlage GmbH*, 2008.

Goodfellow, Ian u.a.: Deep Learning. Das umfassende Handbuch. Frechen: *mitp Verlags GmbH & Co. KG*, 2018.

Jänich, Klaus: Lineare Algebra. 6. Auflage. Berlin/Heidelberg: *Springer-Verlag*, 1996.

Kinnebrock, Werner: Neuronale Netze. Grundlagen, Anwendungen, Beispiele. München: *R. Oldenbourg Verlag GmbH*, 1992.

Krumay, Barbara: Zum Nutzen der Menschheit. Ethische Aspekte und Responsible Research. In: iX Developer. Machine Learning. 6.12.2018, S. 142-146.

LeCun, Yann u. a.: The MNIST database of handwritten digits. 14.5.2013. http://yann.lecun.com/exdb/mnist/ [Zugriff: 25.1.2020].

Rashid, Tariq: Neuronale Netze selbst programmieren. Ein verständlicher Einstieg mit Python. Heidelberg*: O'Reilly Media Inc.*, 2017.

Ribeiro u.a.: "Why Should I Trust You?": Explaining the Predictions of Any Classifier. 16.2.2016. https://arxiv.org/abs/1602.04938 [Zugriff. 2.2.2020].

Scholtysek, Sebastian: Die Robotergesetze von Isaac Asimov. 9.2.2015. http://www.roboterwelt.de/magazin/die-robotergesetze-von-isaac-asimov/ [Zugriff: 2.2.2020].

Spiekermann, Sara: Künstliche Intelligenz – mehr Fluch als Segen. 28.11.2016. https://www.derstandard.at/story/2000048339674/kuenstliche-intelligenz-mehr-fluch-als-segen [Zugriff: 2.2.2020].

Weisstein Eric W.: Vector Direct Product. 2.1.2020. http://mathworld.wolfram.com/VectorDirectProduct.html [Zugriff: 3.2.2020].

Tilly, Marcel: Handwerk hat eine Zukunft. Künstliche Intelligenz – zwischen Hype und Realität. In: iX Developer. Machine Learning. 6.12.2018, S. 138-140.

Abbildungsverzeichnis

Anhang

Matrizen

Definition

Eine Matrix ist eine Rechteckige Anordnung von Elementen nach folgendem Schema (vgl. Beutelsbacher, 2010, S. 50f.):

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

Skalare Multiplikation

Die skalare Multiplikation von Matrizen funktioniert ähnlich wie bei Vektoren (vgl. Beutelsbacher, 2010, S. 51).

$$k * \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} = \begin{pmatrix} k * a_{11} & k * a_{12} \\ k * a_{21} & k * a_{22} \\ k * a_{31} & k * a_{32} \end{pmatrix}$$

Matrix-Vektor-Multiplikation

Das Produkt einer Matrix-Vektor-Multiplikation ist ein Vektor. Dieser beinhaltet die Skalarprodukte des Vektors mit jeder Zeile der Matrix. (Jänich, 1996, S. 89f.)

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \end{pmatrix}$$

Transponierte Matrix

Beim Transponieren einer Matrix werden die Spalten mit den Zeilen getauscht (vgl. Beutelsbacher, 2010, S. 191).

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}^T = \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \end{pmatrix}$$

Weiterführende Rechenregeln für Vektoren

Elementweise Multiplikation

Bei der elementweisen Multiplikation werden die einzelnen Elemente der Vektoren multipliziert.

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \odot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} x_1 y_1 \\ x_2 y_2 \\ x_3 y_3 \end{pmatrix}$$

Dyadisches Produkt

Das dyadische Produkt wird folgendermaßen definiert (vgl. Weisstein, 2020):

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \otimes \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} x_1 y_1 & x_1 y_2 & x_1 y_3 \\ x_2 y_1 & x_2 y_2 & x_2 y_3 \\ x_3 y_1 & x_3 y_2 & x_3 y_3 \end{pmatrix}$$

Quellcode

```
#!/usr/bin/env python
#!/usr/bin/python
#!python
# -*- coding: utf-8 -*-

"""
Autor: Elias Krainer
Version: Python 3.8.1

Dieses Programm wurde im Rahmen der vorwissenschaftlichen Arbeit "Funktionsweise
künstlicher neuronaler Netzwerke: Grundlagen und Programmierung eines
Beispiels" erstellt. Es ist um die Klasse "KNN" gebaut, diese beschreibt ein
mehrlagiges künstliches neuronales Netz auf Basis der Sigmoidfunktion. Es lernt
über ein Gradientenabstiegsverfahren auf Basis von Backpropagation.

Inhalt:
Klasse ANN: Hauptklasse, beinhaltet alle wesentlichen Funktionen eines KNN
Funktion format_mnist: Zum formatieren von MNIST Dateien im "Pickle" Format
Funktion main: Zum Aufrufen der zuvorgenannten Funktionen.

Info: Die Funktion "main" wird aufgerufen wenn das Script als Hauptmodul
ausgeführt wird.
"""

# Drittanbieter Module
import numpy as np # für das Erstellen und Rechnen mit Matrizen


class ANN(object):
    """
    Klasse zur Erstellung eines künstlichen neuronalen Netzes.
    """


    def __init__(self, structure):
        """
```

```
        Konstuktor der Klasse ANN. Hier werden die Gewichtsmatrizen des Netzes
        erstellt.

        Input: structure [list]
        Diese Liste beinhaltet die Anzahl der Neuronen pro Schicht inklusive
        der Eingabeschicht.
        Bsp: [4,5,3] Erstellt ein zweilagiges Netz mit einer Eingabeschicht
        mit 4 Neuronen, eine Versteckte Schicht mit 3 und eine Ausgabeschicht
        mit 3 Neuronen.

        Info: Die Startwerte für die Gewichtsmatrizen werden mit
        normalverteilten Zufallszahlen besetzt, wobei die Standardabweichung
        über folgende Formel berechnet wird: 1/squareroot(n) n ist die Anzahl
        der Neuronen in der vorgeschalteten Schicht.
        """
        self.weights = list() # Liste aller Gewichtsmatrizen als Attribut
        # target: Anzahl der Neuronen in aktueller Schicht
        # previous: Anzahl der Neuronen in zuvorgeschaltener Schicht
        for previous, target in zip(structure[1:],structure[:-1]):
            # eine Matrix mit oben beschriebenem Inhalt wird erstellt. Ihre
            # Größe setzt sich einereits aus der Anzahl voriger Neuronen, sowie
            # der Anzahl an Neuronen in aktueller Schicht +1 zusammen. Diese
            # zusätzliche Reihe kommt daher, dass der Bias ein Teil der Matrix
            # ist.
            weight_matrix = np.random.normal(0, 1/np.sqrt(previous),
                                             (target+1, previous))
            self.weights.append(weight_matrix)


    def sigmoid(self, x):
        """
        Sigmoid Funktion;
        Input: x [number oder np.array]
        """
        return 1 / (1 + np.exp(-x))


    def run(self, activations):
        """
        Einfaches durchlaufen des Netzes. \n
        Input: activations [np.array] \n
        Output: Ausgabe [np.array]
        """
        # activations: Eingabe der aktuellen Schicht
        # weight_matrix: aktualle Gewichtsmatrix
        for weight_matrix in self.weights:
            # Eine 1 wird an nullter Stelle der Eingabe (activations) angehängt
            # aufgrund der Biasspalte der Matrix
            activations = np.insert(activations,0,1)
            # Die übliche Formel zur Berechnung der Ausgabe einer Schicht
```

```
            # sigmoid(a*W)
            activations = self.sigmoid(np.dot(activations, weight_matrix))
        return activations


    def eval_accuracy(self, data):
        """
        Testen der Genauigkeit des Netzes

        Input: data [list] Bsp.: [[Eingabe, Soll-Werte], [...], ...]

        Output: p [float];
        p: rellatiever Anteil der richtig klassifizierter Daten

        Info: Als richtig Klassifiziert gilt, wenn der Index des größten
        Elementes der Ausgabe gleich dem Index des größten Elementes der
        Sollwerten ist.
        """
        correct = 0 # Anzahl richtig klassifizierter Daten
        # activations: Einage, expectations: Sollwerte
        for activations, expectations in data:
            # np.argmax: Index des größtem Element im Vektor
            if np.argmax(self.run(activations)) == np.argmax(expectations):
                correct += 1
        return correct / len(data) # Berechnung des relativen Anteiles


    def backprob(self, activations, expectations):
        """
        Backpropagation Algorithmus für ein Trainingsbeispiel. \n
        Input: activations [np.array]; expectations [np.array] -
        Eingabe sowie Sollwerte des Netzes \n
        Output: gradient [list] - Liste der Gradienten der Gewichtsmatrizen
        """
        # Gleich wie in ANN.run nur, dass alle Eingabewerte in past_activations
        # gespeichert werden
        past_activatios = []
        for weight_matrix in self.weights:
            activations = np.insert(activations,0,1)
            past_activatios.append(activations)
            activations = self.sigmoid(np.dot(activations, weight_matrix))
        past_activatios.append(np.insert(activations,0,1))

        # gradient: Liste aller Gewichtsmatrizen-Gradienten
        gradient = list()
        # error: Teilfehler; layer: aktuelle Schicht, von hinten nach vorne
        # Berechnung des Teilfehlers in letzter Schicht:
        error = (activations - expectations) * activations * (1-activations)
        for layer in range(len(self.weights), 0, -1):
            # Berechnung des Gewichtsmatrizen-Gradienten
```

```
            gradient.append(np.outer(past_activatios[layer-1], error))
            # Berechnung des Teilfehlers der zuvorgeschaltenen Schicht, wobei
            # über das [1:] der Bias weggenommen wird da dieser hier nicht von
            # Bedeutung ist
            error = np.dot(error , np.transpose(self.weights[layer-1][1:])) \
             * past_activatios[layer-1][1:] * (1-past_activatios[layer-1][1:])
        return gradient[::-1] # Neusotierung der Liste (von hinten nach vorne)


    def learn(self, training_data, test_data, epochs, learning_rate ):
        """
        Lernen der Trainingsdaten

        Input: \n
        training_data [list] - Trainingsdaten -
        Bsp. [[Eingabe, Soll-Werte], [...], ...] \n
        test_data [list] - Testdaten -
        Bsp. [[Eingabe, Soll-Werte], [...], ...] \n
        epochs [integer] - Anzahl der Epochen \n
        learning_rate [float] - Lernrate
        """
        for epoch in range(epochs): # Wiederholt sich für jede Epoche
            # Schreibt aktuelle Epoche in den Terminal
            print("epoch %i/%i" % (epoch + 1, epochs))
            np.random.shuffle(training_data) # Trainingsdaten werden gemischt
            # activations: Einage, expectations: Sollwerte
            for activations, expectations in training_data:
                # weight_update ist der Gradient
                weight_update = self.backprob(activations, expectations)
                # Gewichtsmatrizen werden über folgende Formel aktuallisiert:
                # W_neu = W_alt - Lernrate * Gradientenmittelwert
                for n in range(len(self.weights)):
                    self.weights[n] -= weight_update[n] * learning_rate
            # Ausgabe der Genauigkeit
            print("accuracy:", self.eval_accuracy(test_data))


    def feed_backward(self, output_neuron):
        """
        Rückwärtsabfrage des Netzes \n
        Input: output_neuron [integer] - Index des Ausgabeneurons welches für
        die Rückwärtsabfrage aktiv ist. \n
        Output: Ausgabe [np.array] - Ausgabe der Rückertsabfrage in Form
        der Eingabe des Netzes.
        """
        inverse_sigmoid = lambda x : -np.log(1/x-1) # inverse Sigmoidfunktion
        activations = np.full(10, 0.01) # Alle Ausgabeneuronen sind inaktiv
        activations[output_neuron] = 0.99 # nur das deklarierte nicht
        # weight_matrix: aktuelle Gewichtsmatrix (von hinten nach vorne)
```

```
        for weight_matrix in self.weights[::-1]:
            # Ausgabe wird nach vor geleitet
            activations = np.dot(inverse_sigmoid(activations),
             np.transpose(weight_matrix[1:]))
            # Ausgabe wird auf 0-1 skaliert
            activations -= np.min(activations)
            activations = activations / np.max(activations) * 0.98 + 0.01
        return activations



def format_mnist(file):
    """
    Importieren und formatieren der MNIST Daten. \n
    Input: file [string] - Dateipfad \n
    Achtung: Die Daten müssen im Pickle Format vorliegen und alle Bilder als
    "images" und alle Beschriftungen als "labels" in die Datei gespeichert
    worden sein.
    """
    data = np.load(file, allow_pickle=True) # Importieren der Datei
    labels = data["labels"] # extrahieren der Beschriftungen
    images = data["images"] # extrahieren der Bilder
    # skalieren der Grauwerte der Bilder auf 0.01 - 1
    images = images * (0.99 / 255) + 0.01
    formated = list() # Liste der formatierten Dateien
    # image, label : Aktuelles Bild und aktuelle Beschriftung
    for image, label in zip(images, labels):
        # Beschriftungen werden zu Soll-Werten
        # Bsp. 2 -> (0.01, 0.01, 0.99, 0.01, 0.01, 0.01, ...)
        new_label = np.full(10, 0.01)
        new_label[int(label)] = 0.99
        formated.append((image, new_label))
    return formated



def main():
    print("import data")
    # Importieren und Formatieren der Daten
    test_data = format_mnist("mnist_test.npz")
    training_data = format_mnist("mnist_train.npz")
    print("start learning process")
    Net = ANN((784, 50, 50, 10)) # Initialisieren des Netzes
    Net.learn(training_data, test_data, 5, 0.1) # Lernprozess
    print("finished")
```

```python
# Folgende Bedinnung ist wahr, wenn das Script als Hauptmodul ausgeführt wird
# also nicht wenn es als Bibliothek verwendet wird
if __name__ == "__main__":
    main()
```