Neuronale Netze: Grundlagen und einfaches Anwendungsbeispiel

Seminararbeit für das wissenschaftlich-propädeutische Seminar „Angewandte Informatik“

Simon Johann Romedi Morpheus Cyrani

5. November 2018

Inhaltsverzeichnis

1 Einleitung

Es sind gerade einmal wenige Jahrzehnte vergangen, seit Computer ein nicht mehr wegzudenkender Bestandteil unseres alltäglichen Lebens sind. Eine wichtige Rolle hierbei hat offensichtlich die Disziplin der Informatik zusammen mit ihren beteiligten Forschern gespielt, die selbst heute noch etliche Fortschritte präsentieren können. Anfangs konnten die ehemals noch „Rechenmaschinen“ genannten Computer nur einfache Berechnungen durchführen. So brauchte der *Zuse Z3* aus dem Jahre 1941, der als erste digitale Rechenmaschine gilt, beispielsweise ca. drei Sekunden für eine einfache Multiplikation - ganz abgesehen vom nötigen Platzbedarf eines ganzen Raumes (vgl. *Zuse Z3* 2018). Schnell jedoch verbesserte sich die Leistung dank ergiebiger Forschung so stark, dass heutzutage ein einfacher Taschenrechner, den wohl jeder wenigstens in Form einer App auf dem Smartphone hat, nur Bruchteile einer Sekunde für eine Multiplikation braucht, ganz zu schweigen von der Fähigkeit, weitaus kompliziertere Berechnungen durchzuführen. Will man einen genaueren Vergleich, sollte man sich moderne Supercomputer wie den chinesischen *Sunway TaihuLight* aus dem Jahre 2016 anschauen. Dieser ist zwar um die zwanzig Mal größer, hat dafür aber eine Leistungsfähigkeit, die mehrere Billiarden mal so groß wie die des Z3 ist - und das nach nur 75 vergangenen Jahren[1] (vgl. *Sunway TaihuLight* 2018). Man erkennt schnell, welches Potenzial solche Supercomputer vorweisen können: In der Tat werden sie insbesondere für hochgenaue Simulationen in der Chemie, Physik, vom Klima und vielem mehr eingesetzt (vgl. *Supercomputer* 2018). Wenn es also bereits möglich ist, etwas zu berechnen, zu dem ein Mensch keineswegs selbst imstande ist, dann sollte es doch wohl auch ein Leichtes sein, einen Menschen bzw. seine Denkstrukturen selber zu simulieren, oder?
Diese Frage, die verallgemeinert als Forschungsgebiet der *Künstlichen Intelligenz* - kurz: KI - bekannt ist, beschäftigt Forscher tatsächlich schon längere Zeit. Doch entgegen der angenommenen Erwartung steckt dieses noch relativ in den Kinderschuhen. Es existieren zwar schon Systeme, die bestimmte Fähigkeiten des Menschen in einem begrenztem Umfang unter bestimmten Rahmenbedingungen übernehmen können, wie einfache Gespräche führen oder selbstständiges Autofahren, Gehen usw. Aber von einem Gesamtsystem, das alle Funktionen so vereint, dass es einem Menschen Konkurrenz machen könnte, ist man noch weit entfernt. Nichtsdestotrotz deuten die bereits erreichten Fortschritte mit eben genannten Systemen an, dass es sich hierbei um eine äußerst zukunftsträchtige Thematik handelt. Dabei ist das Potenzial so groß, dass sich Durchbrüche darin ebenfalls so gravierend auf uns auswirken können wie die damalige Errungenschaft digitaler Berechnung. Genau deswegen will ich mich in dieser Seminararbeit mit einem der Teilgebiete der KI beschäftigen, nämlich der Umsetzung von

[1] Der Vergleich beruht auf der Einheit *FLOPS*, auf die hier nicht näher eingegangen werden soll.

Assoziationsfähigkeiten und Mustererkennung, wie sie auch bei Menschen vorhanden ist.
Zur genaueren Erläuterung soll das bereits angefangene Paradoxon bezüglich Leistungsfähigkeit von Computern näher ausgeführt werden: Wie ist es einem Menschen möglich, beliebige Objekte mit so hoher Geschwindigkeit als ebensolche identifizieren zu können, auch wenn diese oftmals völlig verschieden aussehen? Warum kann man beispielsweise einen Menschen im Bruchteil einer Sekunde auch als Menschen und nicht als etwas anderes erkennen, trotz der Unmenge an individuellen Merkmalen, und insbesondere auch dann, wenn man nur einen Teil von ihm sieht? Denn genau das ist es, was ein Computer heutzutage noch nicht kann, obwohl er weitaus schneller Berechnungen durchführen kann als ein Mensch. Tatsächlich liegt das Geheimnis dahinter, dass das Gehirn „Daten“ parallel verarbeiten kann. Dieses erkennt quasi alle einzelnen Merkmale gleichzeitig und kann die Spezifikationen eines einzelnen Merkmals auch synchron überprüfen, wodurch es schneller zu einem Ergebnis kommt. Ein konventioneller Computer hingegen kann nur seriell arbeiten, weshalb er alle Möglichkeiten nacheinander abarbeiten muss und sich die somit benötigte Zeit drastisch erhöht. (vgl. Kruse, 2011, 4f.) Um wieder allgemeiner zu werden: Diese Unfähigkeit, dass ein Computer nicht dieselbe Assoziationsfähigkeit wie der Mensch besitzt, ist genau ein Teilgebiet der KI-Forschung. Und ein möglicher Lösungsansatz gründet auf folgender Idee: Was wäre, wenn man die Funktionsweise des Gehirns nachstellen würde? Diese ziemlich biologisch anmutende Herangehensweise basiert auf dem simplen Grundkonzept, dass es genügt, ein Gehirn als ein Netzwerk aus miteinander verknüpften Neuronen anzusehen. Daraus leitet sich ab, ein sogenanntes *künstliches neuronales Netzwerk* programmieren zu können, welches das finale Ziel meiner Arbeit bilden soll.

Im Rahmen dieser Arbeit möchte ich zum Erreichen dieses Zieles die grundlegende Funktionsweise dieser Netzwerke ausführlich darzulegen. Insbesondere sollen verwendete (mathematische) Konzepte, sofern nötig, ebenfalls erklärt und Herleitungen Schritt für Schritt nachvollzogen werden. Anschließend sollen einige Varianten von solchen Netzwerken vorgestellt werden und deren Vor- und Nachteile auf theoretischer und am Ende durch ein eigens programmiertes neuronales Netz auch auf angewandter Ebene verglichen werden.

2 Modellierung

Das erste Kapitel dieser Arbeit befasst sich, wie bereits angedeutet, mit dem Aufbau künstlicher neuronaler Netzwerke. Dazu ist es notwendig, zunächst die „Grundbausteine“, also die Neuronen, sowohl funktionell als auch mathematisch zu beschreiben. Dabei soll stets der Bezug zum biologischen Vorbild gewahrt werden. Im Anschluss daran wird die Gesamtheit der Neuronen betrachtet, die verschiedene Vernetzungen annehmen können und somit auch verschiedene Rollen übernehmen. All das soll das Ziel haben, die Prozesse eines neuronalen Netzes nachvollziehen und am Ende selber als Programm umsetzen zu können.

2.1 Das Neuron

2.1.1 Hintergrund

Zu allererst sollte man einen Blick auf die echten Neuronen richten: Diese bestehen im Allgemeinen aus vier Teilen, nämlich mehrerer Dendriten und Synapsen, einem Zellkörper sowie einem verzweigten Axon. Der Vorgang der Informationsverarbeitung basiert nun darauf, das elektrische Signale versendet werden, welche wiederum andere Signale auszulösen. Konkreter bedeutet das, dass ein Neuron über seine Dendriten Inputsignale erhält, die von anderen Neuronen versendet wurden. Daraufhin versendet es, sofern wenigstens eine minimale, im Zellkörper festgelegte Gesamtspannung erreicht wurde, ein eigenes Signal über sein Axon, dass aufgrund seiner Verzweigung mehrere andere Neuronen bzw. deren Dendriten erreichen kann. Die Synapsen spielen dabei die Rolle eines Verbindungsstücks, welche noch einmal das Signal unterschiedlich abschwächen. Je häufiger die Synapse dabei beansprucht wird, desto weniger schwächt sie das Signal ab, da sie ihre Leitfähigkeit verbessert. Lapidar ausgedrückt verleiht die Synapse dem Signal also eine Gewichtung, welche umso größer ist, je häufiger diese Synapse beansprucht wird, da dies offensichtlich gleichwertig mit einer hohen Relevanz des Signals ist (vgl. Ertel, 2013, S. 248–250).

2.1.2 Grundstruktur

Nachdem der biologische Hintergrund nun erläutert wurde, kann man mit der Umsetzung als Modell beginnen. Zuallererst ist es notwendig, die Neuronen untereinander unterscheiden zu können. In diesem Sinne werden die n Neuronen von 1 bis n durchnummeriert. Neuronen besitzen zunächst einen **Output**, welcher durch die Variablen $o_i, o_j \in \mathbb{R}$ dargestellt wird. Hierbei bezeichnet $i, j \in \{1; ...; n\}$ ein jeweils beliebiges Neuron. Eine Verbindung von einem Neuron i zu einem Neuron j gesondert wiederzugeben, ist nicht nötig. Stattdessen wird direkt die **Gewichtung** $w_{ij} \in \mathbb{R}$ eingeführt.

Diese übernimmt die Funktion der Synapsen, wobei davon abweichend zu beachten ist, dass es ebenfalls negative Gewichtungen geben kann, welche destruktive Signale senden. Daraus lässt sich ein Gesamtwert aller Inputs, genauer gesagt die **Netzeingabe** net_j eines Neurons berechnen. Da nun offensichtlich nicht mit Spannungen, sondern mit Zahlenwerten gerechnet wird, kann man diese Netzeingabe einfach als Summe der einzelnen Inputs nehmen. Der Inputwert für ein Neuron j selbst ist hierbei nicht einfach der Outputwert o_i eines (anderen) Neurons i, vielmehr wird dieser noch mit der Gewichtung w_{ij} der Verbindung multipliziert.

Was fehlt, ist die Berechnung des Outputs o_j, auch **Aktivierung** genannt. Das Problem hierbei ist, dass nicht einfach $o_j = net_j$ gilt. Wenn dem so wäre, dann würde mit fortlaufender Berechnung neuer Outputs eine uneinheitliche und damit einhergehend unübersichtliche Größenordnung entstehen. Da sich dies ebenfalls auf den finalen Output auswirkt, wäre es wenn überhaupt nur mit viel Aufwand möglich, die erhaltenen Werte korrekt zu interpretieren. Um das zu beheben, wird die Netzeingabe noch in eine sogenannte Aktivierungsfunktion f_{akt} eingespeist, die im nächsten Abschnitt genauer betrachtet wird. Allgemein genügt es zu wissen, dass die Aktivierungsfunktion die Netzeingabe auf ein begrenztes und somit endliches Intervall projiziert.

Zuletzt stellt sich noch die Frage, was wäre, wenn bestimmte Neuronen beispielsweise durchgehend eine hohe Aktivierung (nahe dem Maximum des Projektionsintervalls) aufweisen und einen bestimmten Wert quasi nie unterschreiten[2]. Dadurch würde ein eigentlich großer Unterschied in der Netzeingabe dieses Neurons sich kaum in der Aktivierung bemerkbar machen, die Aussagekraft des Neurons würde verloren gehen. Die Lösung ergibt sich durch die Einführung des Schwellwertes $b_j \in \mathbb{R}$ (engl.: **„bias“**).[3] Dieser stellt eine Konstante dar, die immer zur bisherigen Netzeingabe dazugerechnet wird und den Netzeingabewert normieren soll. In diesem Fall soll dadurch also eine neutrale bzw. niedrige und nicht bereits hohe Aktivierung ausgelöst werden. Zusammengefasst erhält man folgende Gleichungen (vgl. Lämmel und Cleve, 2008, S. 197–199):

$$net_j = b_j + \sum_{i=1}^{n} o_i \cdot w_{ij} \tag{1}$$

$$o_j = f_{akt}(net_j) \overset{(1)}{=} f_{akt}(b_j + \sum_{i=1}^{n} o_i \cdot w_{ij}) \tag{2}$$

Der Bias ist vergleichbar mit der Verschiebung von Graphen. Anschaulich kann man sich dazu eine beliebige Parabel der Form $ax^2 + bx + c$ denken. Ändert man nur c, so entspricht dies einer Verschiebung auf der x-Achse. Während also der Verlauf

[2] analog dazu: stets negative Aktivierung (und viele andere Fälle)

[3] Streng genommen ist dies ein spezielles Inputneuron, das ist aber nicht weiter wichtig.

des Graphen gewahrt wird, ändert sich trotzdem der y-Wert (der „Output“) für einen gleichbleibenden x-Wert (das “Output-Gewichtungs-Produkt“).

Was noch zu erwähnen ist, sind die sogenannten **Inputneuronen**. Wie in Abschnitt 2.2.3 das erste Mal zu sehen sein wird, bilden sie die „Startpunkte“ eines Netzes und speisen Daten ein. Dadurch entspricht ihr Output grundsätzlich direkt dem Input (also den Daten) und es kommt zu keiner speziellen Berechnung. (vgl. Kroll, 2013, S.225)

2.1.3 Aktivierungsfunktionen

Zur Vollendung der Beschreibung fehlt noch die zuvor erwähnte Aktivierungsfunktion. Der Grund dafür, diese in einem eigenen Unterkapitel abzuhandeln, ist die Tatsache, dass es mehrere Möglichkeiten gibt, die jeweils verschiedene Ergebnisse liefern können. Deswegen werden im Folgenden die wichtigsten dargestellt, wobei im Sinne der Übersichtlichkeit fortan diese Vereinfachung genutzt wird:

$$x \coloneqq b_j + \sum_{i=1}^{n} o_i \cdot w_{ij} \overset{(1)}{=} net_j$$

Das erste Beispiel soll die ursprüngliche Situation widerspiegeln, bei der die Aktivierungsfunktion zunächst weggelassen wurde. Dies ist gleichzusetzen mit der Aussage, die Identität zurückzugeben, also $f_{Id}(x) = x$, welche wiederum Teil der Gruppe der linearen Funktionen ist. Zusammen mit dem reellen Vorfaktor/der Steigung $m > 0$ beschreibt sich dies wie folgt mathematisch:

$$f_{linear}(x) = m \cdot x \tag{3}$$

Eine Veranschaulichung davon existiert in Abbildung 1. Der Vorteil von solchen Funktionen ist, dass sie simpel und differenzierbar (siehe unten) sind. Hingegen der große Nachteil ist das Fehlen einer Schranke für große und kleine Werte, was aus zuvor genannten Gründen diese Funktion grundsätzlich nicht sinnvoll erscheinen lässt (vgl. Lämmel und Cleve, 2008, S. 198)[4]. Deswegen werden nun Funktionen beschrieben, die eben diese Eigenschaft einer Schranke bzw. eines Grenzwertes aufweisen.
Die einfachste Variante stellt dabei die **Schwellwertfunktion** dar, die ab einem Schwellwert θ für die Eingabe x den Wert 1, ansonsten 0 zurückgibt[5]. Mathematisch ausgedrückt:

$$f_{Schwellwert}(x) = \begin{cases} 1 & \text{für } x \geq \theta \\ 0 & \text{für } x < \theta \end{cases} \tag{4}$$

[4]Quelle wurde verallgemeinert.
[5]Dies sind die üblichsten Rückgabewerte, natürlich könnte man auch andere Zahlen benutzen (häufig ist noch -1 statt 0).

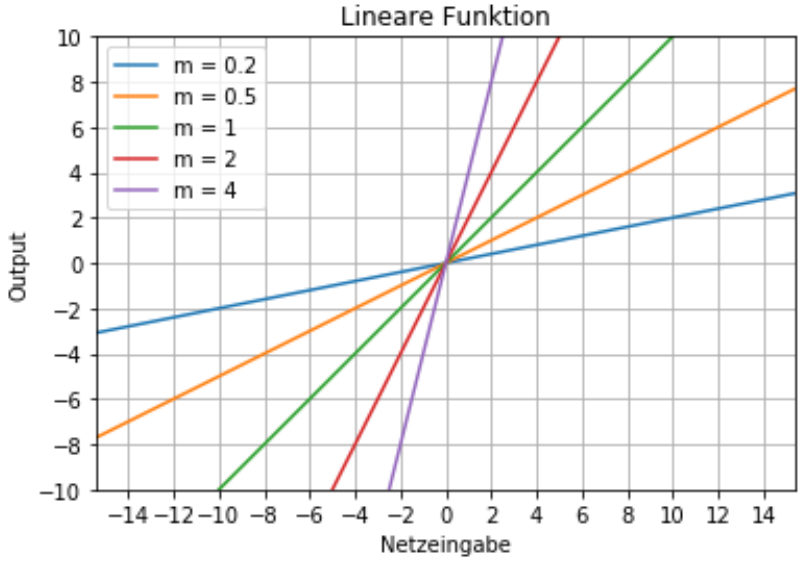


Abbildung 1: Plot linearer Funktionen (eigene Darstellung; siehe Anhang A.1)

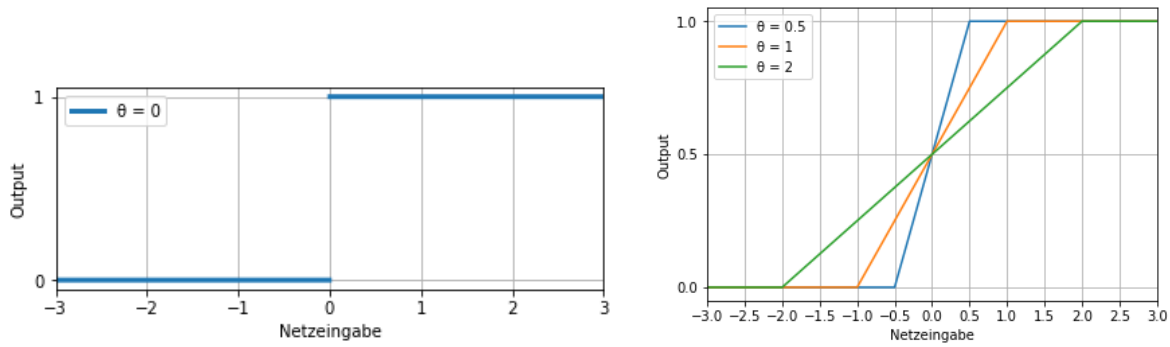
Es handelt sich hierbei also um eine binäre, unstetige Aktivierung, d.h. entweder ist das Neuron vollständig „an“ oder „aus“, wobei der Übergang sprunghaft ist. (vgl. ebd.) Eine andere Möglichkeit, welche (3) mit (4) kombiniert, nennt sich **stückweise lineare Funktion** und ist wie folgt definiert:

$$f_{teil\_linear}(x) = \begin{cases} 1 & \text{für} \quad x > \theta \\ \frac{1}{2} + \frac{x}{2\theta} & \text{für} \quad -\theta \leq x \leq \theta \\ 0 & \text{für} \quad x < -\theta \end{cases} \tag{5}$$

Es gibt hier wieder einen Schwellwert θ, mit dem Unterschied, dass der negative Schwellwert den Bereich für die Aktivierung 0 und der positive Schwellwert analog den Bereich für die Aktivierung 1 festlegt. Dazwischen gibt es eine lineare Steigung (daher der Name), die somit auch „Teilaktivierungen“ zulässt[6]. Diese Funktion ist zwar stetig, aber trotzdem nicht differenzierbar. Man betrachte hierzu auch Abbildung 2. Sinnvoll eingesetzt werden können diese beiden Funktionen in einfachen Netzen, die keine komplexen Aufgaben übernehmen müssen. Dabei wird Funktion (5) für differenziertere Betrachtungen eingesetzt, bei denen ein Übergangsbereich zwischen „an“ und „aus“ sinnvoll erscheint. (vgl. Kruse, 2011, S. 58)

Die folgenden zwei Varianten, die zu den **Sigmoid-Funktionen** gehören, lösen das Problem der Differenzierbarkeit: Die **logistische Funktion**, die zudem auch die wahr-

[6]Dies ist eine unabhängig von der Quelle verallgemeinerte Darstellung. Üblich ist $\theta = \frac{1}{2}$, wodurch sich $\frac{1}{2} + \frac{x}{2\theta}$ zu $\frac{1}{2} + x$ vereinfacht, was der Quelle entspricht, sofern in dieser der *bias* noch eingerechnet wird.



(a) Schwellwertfunktion (b) teilweise lineare Funktion

Abbildung 2: Plot der nicht differenzierbaren Funktionen (eigene Darstellungen)

scheinlich am häufigste verwendete Aktivierungsfunktion ist, wird mit

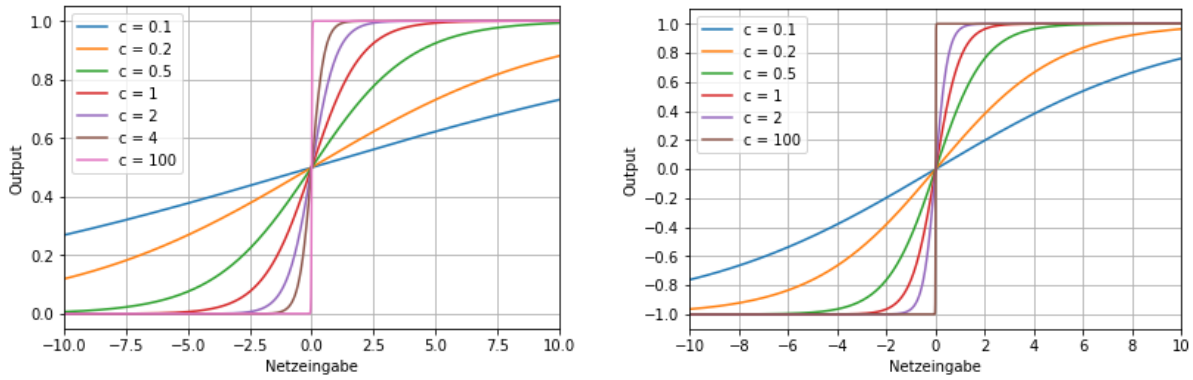
$$f_{log}(x) = \frac{1}{1 + e^{-c \cdot x}} \tag{6}$$

beschrieben. Hierbei ist e die *eulersche Zahl* und es gilt für den reellen Parameter $c > 0$. (vgl. Lämmel und Cleve, 2008, 198f.) Wie man schnell nachprüfen kann, ist $\lim_{x \to -\infty} f_{log}(x) = 0$ sowie $\lim_{x \to +\infty} f_{log}(x) = 1$.[7] Das Besondere ist hierbei, dass mit c die Geschwindigkeit dieser Annäherung festgelegt werden kann. Dabei gilt, je größer c ist, desto schneller geschieht dieser Umbruch von 0 auf 1, was man gut in Abbildung 3a erkennen kann. Insbesondere gilt deshalb $\lim_{c \to +\infty} f_{log}(x) \approxeq f_{Schwellwert}(x)$ mit $\theta = 0$. (vgl. Lämmel und Cleve, 2008, 198f.)
Eine Alternative mit -1 als unteren Grenzwert ist durch einen modifizierten **Tangens Hyperbolicus** (siehe Abbildung 3b) gegeben, der folgendermaßen definiert ist:

$$f_{tanh}(x) = 1 - \frac{2}{e^{2x \cdot c} + 1} \tag{7}$$

Hierbei gilt für c gleiches wie zuvor[8]. Durch die Möglichkeit, dass die Funktion auch negative Werte liefern kann, können einfacher destruktive Signale versendet werden. (vgl. ebd. sowie *Tangens hyperbolicus und Kotangens hyperbolicus* 2018)
Der große Vorteil von (3) sowie den sigmoiden Funktionen (6) und (7) als Aktivierungsfunktionen ist die Tatsache, dass diese differenzierbar sind. Denn daraus folgt unmittelbar, dass diese ableitbar sind, was für einige effektive Lernverfahren, besonders in komplexeren neuronalen Netzen, unabdingbar sein wird (siehe Kapitel 3). Aufgrund komplexerer Rechenoperationen ist dies im Gegenzug jedoch mit höheren Laufzeiten verbunden, sodass es abzuschätzen gilt, ob andere Aktivierungsfunktionen bei geringe-

[7] Wäre $c < 0$, dann wären Null und Eins vertauscht. Wäre $c = 0$, würde sich der Funktionsterm zur Konstanten $\frac{1}{2}$ vereinfachen.
[8] Für $c = 1$ gilt zudem: $f_{tanh} = \tanh(x)$



(a) logistische Funktion

(b) Tangens Hyperbolicus

Abbildung 3: Plot der Sigmoid-Funktionen (eigene Darstellungen)

rem Aufwand nicht ähnlich gute Ergebnisse liefern würden.
Zusammengefasst wäre damit nun die Implementierung von Neuronen als alleinstehende Objekte in einem Programm möglich. Was jedoch fehlt, ist der Aspekt der Neuronenverknüpfung, womit sich der nächste Abschnitt beschäftigt.

2.2 Das künstliche neuronale Netz als Verknüpfung von Neuronen

Neuronale Netze bestehen üblicherweise aus mehreren hundert Neuronen. Diese sind normalerweise nicht beliebig miteinander verbunden, sondern in sogenannten **Schichten** angeordnet. Nach Beschreibung der dafür existierenden Varianten sollen für die simpelste Form mathematische Vereinfachungen hergeleitet werden, die als Basis für die anschließende Beschreibung von den zwei wichtigsten Netzstrukturen dienen sollen. (vgl. Kroll, 2013, S.226)

2.2.1 Überblick der Netztypen

Die Wahl der Netzstruktur spielt eine äußerst essentielle Rolle: So gibt es beispielsweise Netze, deren Neuronen eine Verbindung mit sich selbst, mit Neuronen innerhalb der selben oder aber mit beliebigen Schichten eingehen. Diese sind zusammengefasst unter dem Begriff **rückgekoppeltes Netz** und legen dynamischere Berechnungen[9] an den Tag. Da eine detaillierte Beschreibung solcher Netze aber den Rahmen dieser Arbeit sprengen würde, wird sich hier auf eine grundlegendere Form von Netzen festgelegt, bei denen Neuronen einer Schicht nur Verbindungen zu Neuronen der darauffolgenden Schicht aufweisen.

[9]Das heißt, dass mehr Wechselwirkungen zwischen den Neuronen berücksichtigt werden und vorige Zustände des Netzes miteinbezogen werden.

Diese Netze werden **Feedforward-Netze** genannt, welche sich ganz allgemein dadurch auszeichnen, Dinge *klassifizieren* zu können, d.h. einer Klasse/Gruppe zuzuordnen. Die Klassifizierung geschieht dadurch, Datenmengen anhand Muster in ihren Eigenschaften voneinander zu separieren. (vgl. a.a.O. S.226f.) Für diese Form von Netzen soll nun angesprochene Formalisierung hergeleitet werden.

2.2.2 Mathematische Verallgemeinerung

Eine Unterteilung in m Schichten ermöglicht jedem der n Neuronen eine eindeutige Zuordnung. Dadurch lässt sich anstatt einer globalen Nummerierung der Neuronen eine schichtinterne aufsetzen. Wenn $o_i^{(a)}$ sich nun auf ein Neuron i in Schicht $a \in \{1, ..., m\}$ (analog für j) bezieht, und wenn $n^{(a)}$ die Anzahl der Neuronen in Schicht a referenziert, dann lässt sich folgende, auf Gleichung (2) basierende Notation definieren:

$$o_j^{(a+1)} = f_{akt}(b_j^{(a+1)} + \sum_{i=1}^{n^{(a)}} o_i^{(a)} \cdot w_{ij}^{(a)}) \tag{8}$$

Dies eröffnet die Basis zur allgemeinen Darstellung des Outputs einer Schicht durch Bemächtigung von *Matrizen*[10] und *Vektoren*. Der Grundgedanke besteht darin, einen **Vektor** $o^{(a)}$ so darzustellen, dass dessen i. tes Element den Output vom Neuron i in Schicht a beinhaltet. Analog dazu ergeben sich Vektoren für den Output von Neuron j und den Schwellwert b. Etwas anders lässt sich die Gewichtung w_{ij} zwischen zwei Neuronen aufeinanderfolgender Schichten darstellen: Ausgedrückt durch das Symbol $\mathbf{W^{(a)}}$ bezeichnet sie eine Art Tabelle, in der Mathematik **Matrix** genannt, dessen Elemente mit den Zeilen- bzw. Spaltenkoordinaten i bzw. j die entsprechende Gewichtung beinhaltet.

Mit diesen Feststellungen lässt sich Gleichung (8) zu einem sogenannten *Matrix-Vektorprodukt* mit anschließender Vektoraddition umformen[11] (vgl. Sanderson, 2017, Teil 1):

$$o^{(a+1)} = f_{akt}(\mathbf{W}^{(a)} \cdot o^{(a)} + b^{(a+1)}) \tag{9}$$

Ausgehend von diesen Betrachtungen sollen nun zwei grundlegende Netztypen beschrieben werden.

2.2.3 Einlagiges Perzeptron

Die grundlegendste Form eines Feedforward-Netzes ist ein **einlagiges Perzeptron**. Es besteht aus einer *Inputschicht* und einer *Outputschicht*, die jeweils aus ein bis theo-

[10] Aus Gründen der Kürze wird hier auf eine genaue Erläuterung verzichtet.
[11] Die Aktivierungfunktion wird auf die *einzelnen* Elemente des resultierenden Vektors angewandt.

retisch beliebig vielen Neuronen bestehen. Diese Neuronen werden entsprechend auch als *Inputneuronen* bzw. *Outputneuronen* bezeichnet. Dabei ist die Abgrenzung zu den Begriffen *Input* und *Output* eines Neurons zu beachten: Die Schichtbezeichnung gibt nur an, in welcher Schicht die Daten für das Netz eingespeist werden, und in welcher Schicht die gewünschten Ergebnisse ausgegeben werden, die Funktion der Neuronen bleibt im Kern identisch. Ferner bezieht sich die Bezeichnung *einlagig* darauf, dass nur Verbindungen existieren, die von der Input- zur Outputschicht führen und es somit nur eine Lage an Verbindungen/Gewichtungen gibt. Dementsprechend führen grundsätzlich auch nur die Outputneuronen Berechnungen aus, da die Inputneuronen bloß die Daten einspeisen und unverändert übergeben[12]. (vgl. Ertel, 2013, S. 199)

Diese Daten repräsentieren in Form eines numerischen Wertes pro Inputneuron jeweils eine spezifische Eigenschaft eines beliebigen Objektes. Um darauf basierend eine Unterteilung der Objekte zu gewährleisten, macht das Perzeptron Gebrauch von **linearer Separierung**. (vgl. ebd.) Vorstellen lässt sich dies als ein mehrdimensionales Koordinatensystem[13], dessen Achsen jeweils eine der Eigenschaften durch Zahlen beschreiben. Die Objekte, die bezüglich dieser Eigenschaften beschrieben sind, können dann als Punkte in dieses Koordinatensystem eingetragen werden. Dadurch ist es möglich, den Koordinatenraum einschließlich der darin liegenden Datenpunkte durch Unterräume mit genau einer Dimension weniger[14] zu unterteilen. Dazu gehören *Punkte*, *Geraden*, *Ebenen* oder *Hyperebenen*[15]. (vgl. *Hyperebene* 2018) Jene Unterräume sind somit ferner der jeweils einzig endlose (Vektor-)Raum, der den ebenfalls endlosen Koordinatenraum in mehrere (exakt: 2) Teile zu zerlegen vermag.

Die Anzahl der genutzten Unterräume hängt hierbei mit der Rolle der Neuronen der Outputschicht zusammen: Jedes Outputneuron kann so interpretiert werden, dass es ein Objekt einer spezifischen Klasse zuweist, indem dessen Aktivierung hoch ist, oder umgekehrt durch eine zu geringe Aktivierung nicht zuweist. Dadurch entsteht pro Outputneuron ein Unterraum, definiert durch die Aktivierungsfunktion des Neurons, der die Objekte einer Klasse vom gesamten Rest abtrennt. Ein einfaches Beispiel ist in Abbildung 4 zu sehen. In der Summe folgt daraus eine Abgrenzung aller Klassen voneinander. Mit n Outputneuronen können also über n Unterräume (mindestens) n Klassen unterschieden werden. Wie genau die Bestimmung der Unterräume abläuft, wird in Kapitel 3 erläutert. (vgl. Ertel, 2013, S. 199f.)

Abgesehen davon ist es in diesem Fall durchaus sinnvoll, die Schwellwertfunktion (4) als Aktivierungsfunktion für die Outputneuronen heranzuziehen, da deren Ausgabe als

[12] Es ist trotzdem möglich, dass ein Inputneuron bereits eine Transformation des Wertes durchführen kann, wie es auch im praktischen Teil der Arbeit geschehen wird, aber nicht notwendig.

[13] genauer: ein Vektorraum

[14] genauer: ein Vektorraum, der durch einen Richtungsvektor weniger definiert ist

[15] Hyperebene bezeichnet die analoge Verallgemeinerung des Begriffs der „*Ebene als Unterraum eines dreidimensionalen Koordinatensystems*“ für beliebig höhere Dimensionen.

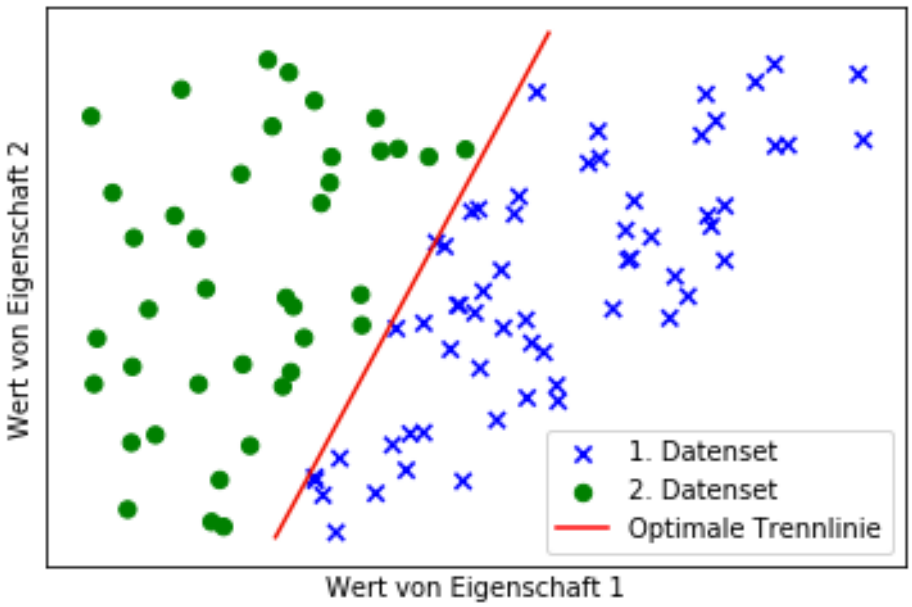


Abbildung 4: Eine Klassifizierung von zwei fiktiven Datenmengen mit zwei Eigenschaften, mit entsprechend zwei Neuronen in der Inputschicht eines Perzeptrons, welche die dargestellte Trennung möglichst gut annähern müssen. Der zweidimensionale Koordinatenraum wird durch die eindimensionale Trennlinie als Unterraum in zwei Teile zerlegt (eigene Darstellung; siehe Anhang A.1)

ein Wahrheitswert interpretiert werden kann. Dies entspricht genau der Zuordnung, ob das Objekt zu einer Klasse gehört, oder nicht. Nichtsdestominder schließt das nicht die Wahl sigmoider Funktionen aus, da sie die Zuordnungssicherheit (auch: **Konfidenzintervall**) des Netzwerkes akkurater repräsentieren können. (vgl. ebd.)

2.2.4 Mehrlagiges Perzeptron

Belässt man das Verfahren aus vorigem Kapitel so wie es ist, ergibt sich ein Problem: Nicht jede Menge an Daten ist ohne Weiteres linear separabel, wie in Grafik 5 veranschaulicht ist. Die Lösung hierfür liefert das **mehrlagige Perzeptron**. Es unterscheidet sich vom einlagigen Perzeptron dadurch, dass es zusätzlich noch ein oder mehrere sogenannte *verdeckte Schichten* beinhaltet, die zwischen Input- und Outputschicht liegen, siehe hierzu auch Abbildung 6. Dies ermöglicht eine nichtlineare Klassifikation, d.h. die Grenzen können „unförmigere“ Gestalten annehmen. Die Funktionsweise basiert darauf, dass jede verdeckte Schicht eine weitere Transformation des ursprünglich linearen Unterraums vornehmen kann, wie z.B. die Position eines Knickes bestimmen, oder mehrere Unterräume zur Abtrennung einer einzelnen Klasse vom Rest zu defi-

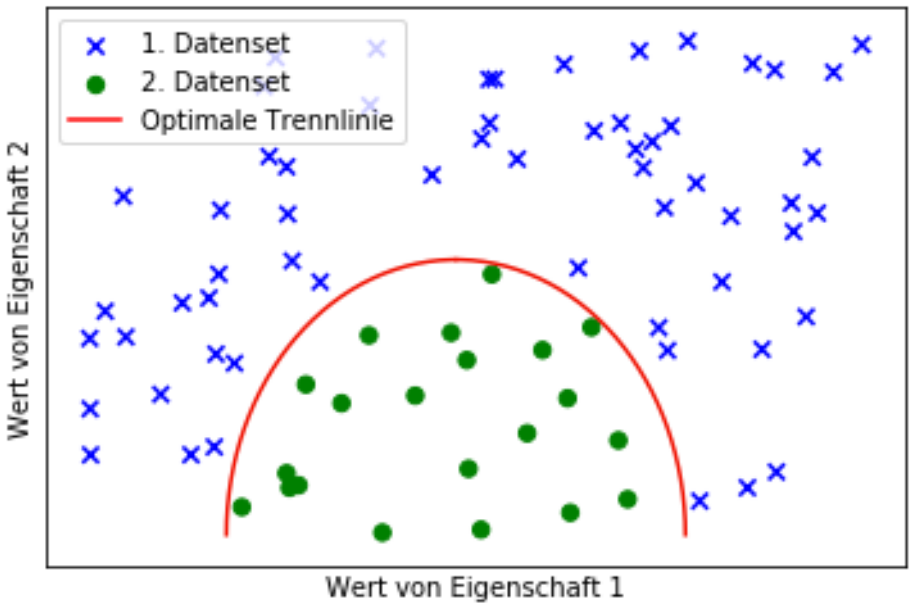


Abbildung 5: Zwei nicht durch Geraden voneinander trennbare, fiktive Datenmengen (eigene Darstellung; siehe Anhang A.1)

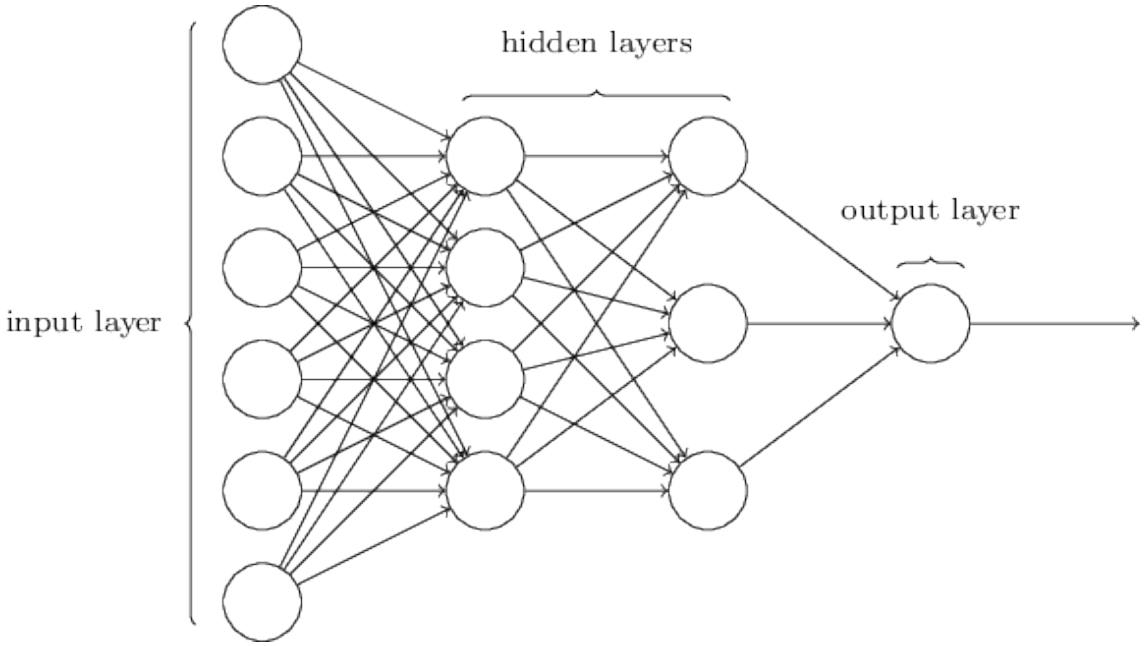


Abbildung 6: Typischer Aufbau eines mehrlagigen Perzeptrons.[a]

[a] Bildquelle:https://raw.githubusercontent.com/ledell/sldm4-h2o/master/mlp_network.png

nieren.[16] Dadurch erst wird eine gute Klassifikation der beispielsweise in Abbildung 5 dargestellten Daten möglich. (vgl. Gableske, 2005, S. 2)

[16] Die exakten Funktionsweisen sind jeweils sehr divers ausgeprägt, weshalb hier auf eine detaillierte Erläuterung verzichtet wird.

3 Lernverfahren

Obwohl es nun möglich wäre, das gesamte neuronale Netz zur Klassifizierung implementieren zu können, ist es dennnoch nicht möglich zu sagen, wie die einzelnen Gewichtungen und Bias gewählt werden sollen. Aus diesem Grund behandelt dieses Kapitel zwei sogenannte **Lernverfahren**, mit denen Veränderungen automatisiert vorgenommen werden sollen. Für den Fall der Feedforward-Netze beschränken sich die Änderungen auf Anpassung der Gewichtung und des Bias. Dazu ist es zunächst notwendig, die zwei Gruppen von Lernverfahren zu unterscheiden:

Überwachtes Lernen zeichnet sich dadurch aus, dass eine gewisse Menge an Trainingsmustern existieren muss, die zunächst auf das neuronale Netz angewendet werden müssen. Darunter versteht man, dass von den Trainingsdaten die erwünschten Outputwerte bekannt sind, sodass basierend auf den Abweichungen des errechneten Outputs Änderungen vorgenommen werden. Nach einmaliger Anwendung einer genügend großen Menge an Trainingsdaten bis Erreichen des gewünschten Grades der Zuverlässigkeit können unbekannte Daten eingesetzt werden. Ab diesem Moment werden grundsätzlich auch nie mehr neue Änderungen vorgenommen, da dass Netz auf ein spezifisches Klassifizierungsproblem hintrainiert wurde. Das hat zwar den Vorteil, dass das Netz schneller arbeitet, weil es zusätzlich keine Änderungen berechnen muss, gleichzeitig muss die Klassifizierung aber statisch bleiben. (vgl. Görz, Schneeberger und Schmid, 2014, S. 375f.)

Ein großes Problem vom überwachten Lernen ist die sogenannte **Überanpassung** bzw. **Unteranpassung**. Bestimmte Netzeigenschaften wie die Anzahl von Neuronen in den verdeckten Schichten können nicht durch Regeln eindeutig festgelegt werden. Stattdessen müssen verschiedene Werte ausprobiert werden. Wenn jedoch zu viele bzw. zu wenige Neuronen verwendet werden, dann kann sich das Netz zu genau bzw. zu ungenau an Trainingsdaten anpassen. Im ersteren Fall, der *Überanpassung*, würde es bei unbekannten Daten schlechter abschneiden als bei den Trainingsdaten, da das Netz kaum Spielraum zulässt. Im letzteren Fall, der *Unteranpassung*, würde es bereits bei der Klassifizierung der Trainingsdaten geringe Genauigkeiten erzielen. Um den Grad der Überanpassung (grob) bestimmen zu können, muss es deswegen immer separate Trainingsmuster geben, die nicht zum Lernen benutzt werden und dadurch dem Netz unbekannt sind. (vgl. ebd.)

Eine Unterform vom überwachten Lernen ist *bestärkendes Lernen*, bei der es zwar keine Trainingsdaten gibt, aber trotzdem das Endergebnis bezüglich seines Erfolgs, ein bestimmtes Ziel zu erreichen, bewertet wird. (vgl. Aunkofer, 2017)

Unüberwachtes Lernen als Gegensatz dazu benutzt keine Trainingsmuster, sondern lernt allein aufgrund der Tatsache der unterschiedlichen Beanspruchung der Neuronen und dem selbstständigen Erkennen von Mustern in der Beanspruchung für verschiedene Daten. Dadurch ist die Anzahl der Klassen normalerweise auch nicht festgelegt, sondern das Netz bestimmt sie selber im Laufe des Prozesses. Das ist nützlich, wenn es beispielsweise zu wenig oder keine Trainingsdaten gibt, oder wenn die Klassen temporären Änderungen unterliegen, sodass das Netz während normaler Benutzung dynamisch angepasst wird. Zudem ist diese Art sinnvoll, wenn anfangs noch unbekannt ist, wie viele Klassen es geben soll. Deswegen ist dies auch näher am biologischen Vorbild (Abschnitt 2.1.1) orientiert, welches genau auf diesen Eigenschaften aufbaut. Der Nachteil ist jedoch die verlängerte Laufzeit, da einerseits ständig zusätzliche Berechnungen zur Netzanpassung durchgeführt werden, andererseits Verfahren dieser Art generell langsamer lernen. Außerdem ist es nicht gegeben, dass das Netzwerk sich in die gewünschte Richtung entwickelt und es nicht ausgeschlossen ist, Dinge nach Kriterien sortiert, die eigentlich schwachsinnig oder für das Problem nicht sinnvoll sind. Dementsprechend werden oft mehrere Anläufe benötigt, um ein geeignetes Netz zu finden.

Das einfachste Beispiel für ein unüberwachtes Lernverfahren bildet die **Hebb-Regel**, bei der aber zu Beginn die Anzahl der Klassen bekannt sein müssen. Andere Lernverfahren, die davon unabhängig sind, benötigen andere Netzstrukturen. Für den Umfang dieser Arbeit genügt es deshalb, nur die überwachten Lernverfahren zu betrachten. (vgl. Görz, Schneeberger und Schmid, 2014, S. 380f)

3.1 Deltaregel

3.1.1 Herleitung

Das erste (überwachte) Lernverfahren, dass vorgestellt wird, nennt sich **Delta-Regel**. Sein Anwendungsbereich beschränkt sich auf einschichtige Perzeptren. Der Kerngedanke zur Funktionsweise basiert auf einer *Änderung der Gewichtungen* Δw_{kj} mit $k \in \{1, ..., n\}$ für ein jedes Trainingsmuster (vgl. Ertel, 2013, S.272):

$$w_{kj}^{neu} = w_{kj}^{alt} + \Delta w_{kj} \tag{10}$$

Es ist zu beachten, dass in der Delta-Regel kein Gebrauch vom Bias gemacht wird, weswegen dieser für dieses Kapitel ignoriert wird. Ferner ist das Ziel der Delta-Regel, einen **Fehler** E für jedes untersuchte Trainingsmuster kleiner werden zu lassen. Dieser

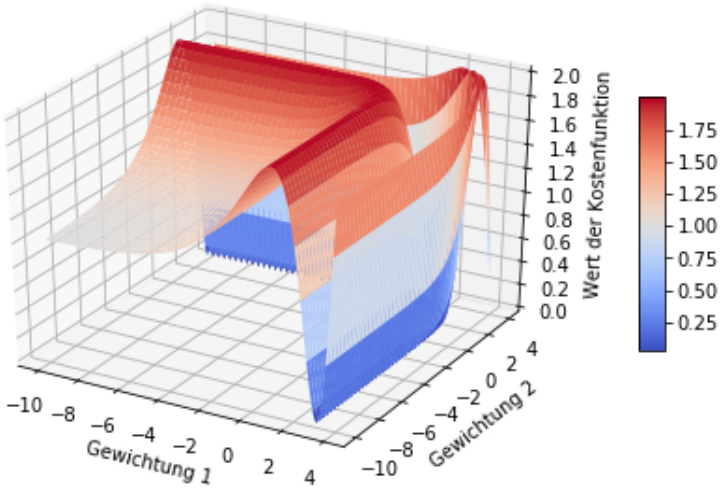


Abbildung 7: Plot einer fiktiven Kostenfunktion, die von zwei Gewichtungen abhängt. Dabei befinden sich an den tiefblauen Stellen die globalen Täler, während der weiße Bereich nur zu einem lokalen Minimum führt (eigene Darstellung; siehe Anhang A.1)

Fehler ist wie folgt definiert (vgl. Kroll, 2013, S.239):

$$E_j = t_j - o_j \tag{11}$$

$$E_{ges} = \frac{1}{2}\sum_{j=1}^{n}(E_j)^2 \overset{(11)}{=} \frac{1}{2}\sum_{j=1}^{n}(t_j - o_j)^2 \tag{12}$$

Hierbei bezeichnet t_j den Zielwert vom Outputneuron o_j für ein Trainingsmuster, wodurch sich der Fehler dieses Outputs E_j aus der Differenz daraus ergibt. Der Gesamtfehler ergibt sich aus der quadrierten Summe der Einzelfehler, damit große Abweichungen einen signifikanten Einfluss haben, während kleine Fehler vernachlässigbar bleiben. Der Koeffizient $\frac{1}{2}$ ergibt sich daraus, dass sich dieser nach dem Ableiten (siehe unten) wegkürzt[17]. Das Ziel ist es zwar, den gesamten Fehler über alle n Outputneuronen zu minimieren, doch da für das einschichtige Perzeptron die Gewichtungen immer zu exakt einem Output zeigen, sind die einzelnen Fehler unabhängig voneinander. Dies ermöglicht wiederum eine separate Betrachtung der Outputneuronen. (vgl. ebd.)

Nun stellt sich die Frage, wie man anhand des Fehlers auf die Änderung der Gewichtungen schließen kann. Folgende Veranschaulichung ist zu diesem Zwecke förderlich: Der Fehler kann als eine Funktion, man sagt auch **Kostenfunktion** angesehen werden, die von den Gewichtungen abhängig ist. Diese Tatsache folgt für den Output o_j direkt aus Gleichung (2). Dadurch wird für jedes Trainingsmuster eine Verbildlichung als Graph möglich, bei dem jede Gewichtung sowie der berechnete Fehler einer Ach-

[17]Das hat vorrangig nur ästhetische Gründe.

se zugeordnet wird. Ein Beispiel liefert Abbildung 7. In diesen Graphen befinden sich sogenannte *Täler*, d.h. Punkte, an denen die Kostenfunktion ein lokales Minimum aufweist. Da diese Minima durch die Werte für die Gewichtungen beschrieben sind, ist das Ziel nun, sich einem dieser Punkte anzunähern. Dabei ist es jedoch unvorteilhaft, einfach ein Tal auszulesen, was vor allem zwei Gründe hat:

Erstens gibt der Graph nur die Kostenfunktion von *einem* Trainingsmuster wieder. Aus diesem Grund würde für jedes Trainingsmuster immer ein neues Minimum gefunden werden und die bisherigen Anpassungen nichtig werden. Das neuronale Netz soll aber alle Muster als zusammengehörige Gruppe klassifizieren, weshalb es essentiell ist, sich einem „durchschnittlichen" Ort der Täler anzunähern. Zudem gibt es immer von der Norm abweichende, deviante Trainingsmuster (auch **Rauschen** genannt), die das gesamte Netz verfälschen würden (vgl. Ertel, 2013, S.195).

Zweitens müsste für jedes Trainingsmuster ein Graph geplottet werden bzw. wenigstens unzählige Punkte berechnet werden. Da diese Anzahl offensichtlich exponentiell zur Anzahl der Gewichtungen wächst, führt dies schnell zu hohen Laufzeiten[18]. Insbesondere ist die Position der Minima nicht ohne Weiteres ersichtlich, sodass teilweise riesige Bereiche abgedeckt werden müssen, bevor ein Minimum gefunden wird.

Die Lösung für dieses Problem ergibt sich daraus, den **Gradienten** ∇ anzuwenden, der wie folgt definiert ist (vgl. a.a.O. S.272):

$$\nabla f(x_1, \cdots, x_n) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix} \tag{13}$$

Dieser liefert für eine Funktion mit gegebenen Werten also einen Vektor, dessen einzelne Elemente die *partielle Ableitung* einer Funktion f nach jeweils einer ihrer Variablen ist.[19] Da die partielle Ableitung die momentane Änderungsrate auf der Achse der abgeleiteten Variablen wiedergibt, muss der Vektor folglich in die Richtung mit der größten Änderungsrate zeigen. Bewegt man sich dementsprechend in Richtung des Gradienten, so wächst der Funktionswert zunächst. Dabei ist zu beachten, dass der Gradient nicht direkt zum Maximum zeigt, weshalb diese Bewegung nur in genügend kleinen Schritten einschließlich ständiger Aktualisierung des Gradienten vonstattengehen kann. Mit dieser Methodik konvergiert der Funktionswert also entweder gegen ein lokales Maximum, oder er steigt ins Unendliche. Nun soll die Kostenfunktion jedoch minimiert, und nicht maximiert werden. Deshalb ist es notwendig, den Gegenvektor des Gradien-

[18] Man bedenke, dass in einem typischen neuronalen Netz zehn bis mehrere hundert Gewichtungen üblich sind.

[19] Die partielle Ableitung einer Funktion $\frac{\partial f(x_1, x_2, \ldots)}{\partial x_i}$ funktioniert im Kern identisch zum einfachen Ableiten, mit dem Unterschied, dass alle anderen Variablen $x_j \neq x_i$ als Konstanten behandelt werden.

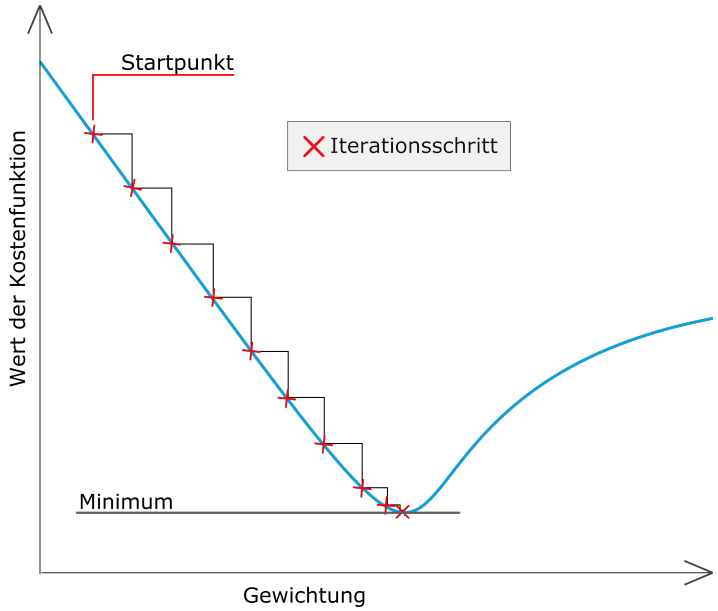


Abbildung 8: Veranschaulichung des Gradientenabstiegsvefahren anhand einer fiktiven Kostenfunktion, die von einer Gewichtung abhängt. (eigene Darstellung)

ten bzw. den Vektor des negativen Gradienten[20] zu nutzen. Analog für das Maximum führt eine Bewegung in Richtung des Vektors des negativen Gradienten Schritt für Schritt zum Minimum, weshalb dieses Verfahren auch **Gradientenabstiegsverfahren** genannt wird[21], siehe Abbildung 8 für ein Beispiel. (vgl. Lämmel und Cleve, 2008, S.211f.)
Hier wird nun auch die Bedeutung der Kostenfunktion ersichtlich, da die Minima durch die Quadrierung nie unter 0 fallen können. Andernfalls bestünde die Möglichkeit, dass der Gradientenabstieg gegen einen negativen Wert, welches betragsmäßig wiederum ein Maximum ist, konvergiert.

Aus dieser Illustration kann man dadurch folgende Gleichung für die Änderung konkreter Gewichte aufstellen (vgl. Kroll, 2013, S.238):

$$\Delta w_{kj} = -\frac{\partial E_{ges}}{\partial w_{kj}} \tag{14}$$

Dabei ist $\frac{\partial E_{ges}}{\partial w_{kj}}$ ein Eintrag im Gradienten. Da es nicht offensichtlich ist, die Fehlerfunktion nach der Gewichtung abzuleiten, bedient man sich zunächst der Kettenregel (vgl. Lämmel und Cleve, 2008, S.213):

$$-\frac{\partial E_{ges}}{\partial w_{kj}} = -\frac{\partial E_{ges}}{\partial o_j}\frac{\partial o_j}{\partial w_{kj}} \tag{15}$$

[20]Zur Erinnerung: Für einen Vektor $\vec{v}$ ergibt $-\vec{v}$ den Gegenvektor.
[21]Tatsächlich weist das Gradientenabstiegsverfahren gewisse Ähnlichkeiten zum allgemein bekannten Newton-Verfahren auf und kann als eine Erweiterung dessen betrachtet werden.

$f(x)$	$f_{lin}(x)$	$f_{log}(x)$[22]	$f_{tanh}(x)$[23]
$f'(x)$	m	$f_{log}(x) \cdot (f_{log}(x) - 1)$	$1 - f_{tanh}(x)^2$

Tabelle 1: Auflistung der Ableitungen für differenzierbare Aktivierungsfunktionen

Dadurch ergibt sich (beachte $b_j = 0$)(vgl. a.a.O. S.213f.):

$$\begin{aligned} -\frac{\partial E_{ges}}{\partial o_j}\frac{\partial o_j}{\partial w_{kj}} &\overset{(12),(2)}{=} -\frac{\partial}{\partial o_j}\frac{1}{2}\sum_{j=1}^{n}(t_j - o_j)^2 \cdot \frac{\partial}{\partial w_{kj}} f_{akt}(\sum_{i=1}^{n} o_i \cdot w_{ij}) \\ &= (t_j - o_j) \cdot f'_{akt}(\sum_{i=1}^{n} o_i \cdot w_{ij}) \cdot o_k \end{aligned} \tag{16}$$

Der Grund, dass die Ableitung der Netzeingabe $\sum_{i=1}^{n} o_i \cdot w_{ij}$ einfach o_k ist, beruht auf der Tatsache, dass nur der Summand $o_k \cdot w_{kj}$ die Gewichtung enthält, nach der abgeleitet wird. Dementsprechend sind alle anderen Summanden Konstanten, und die Ableitung einer Konstante ist 0. Zudem wird hier die in Abschnitt 2.1.3 erwähnte Notwendigkeit der Differenzierbarkeit von Aktivierungsfunktionen ersichtlich: Wenn die Aktivierungsfunktion nicht differenzierbar wäre, dann wäre auch die Delta-Regel nicht anwendbar. Als Folge daraus sind in Tabelle 1 alle genannten Aktivierungsfunktionen zusammen mit ihrer Ableitung aufgelistet.

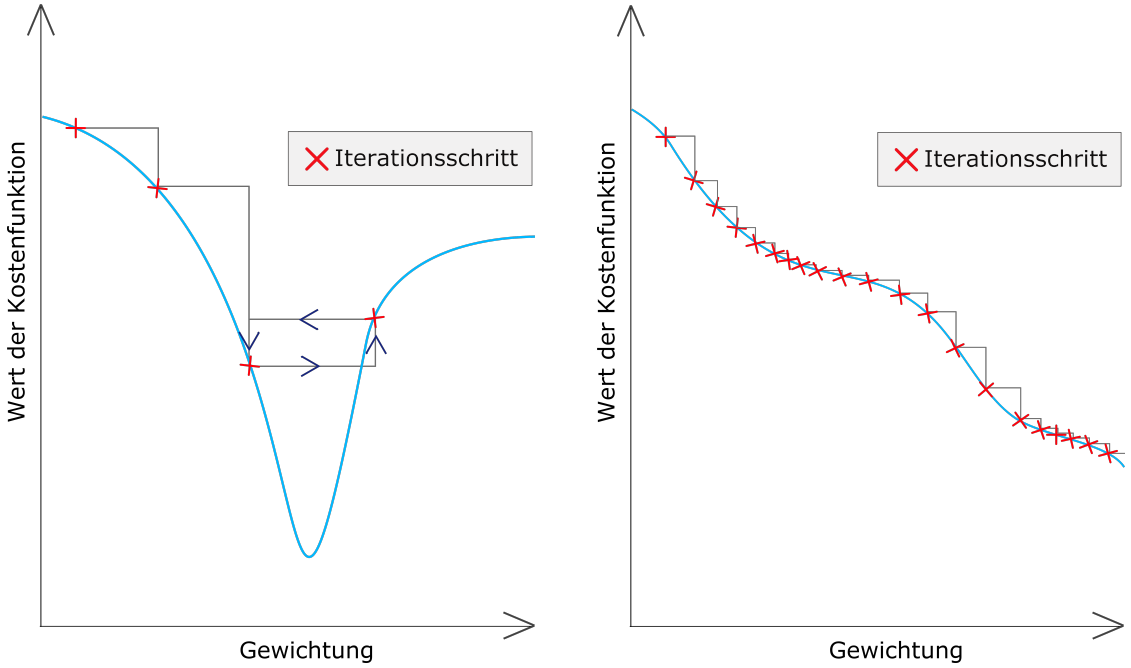
3.1.2 Weitere Maßnahmen

Um damit nun eine Datenmenge zu klassifizieren, darf nicht nach jedem Trainingsmuster eine Gewichtsaktualisierung vorgenommen werden, sondern es muss eine genügend große Menge an Trainingsmustern verwendet werden. Diese unter dem Namen **Batch-Learning** (vgl. Lämmel und Cleve, 2008, S.213) bekannte Vorgehensweise hat zur Folge, dass der durchschnittliche Graph der Mustermenge in etwa konstant bleibt und insbesondere die Minima sich nicht verändern, was wiederum essentiell zum Konvergieren ist, dafür aber mehr Rechenzeit in Abhängigkeit der Größe jeder Menge benötigt. Mathematisch ausgedrückt ist die Gewichtsänderung nach R Mustern ihr Durchschnittswert, wobei r als hochgestellter Index die aktuelle Musternummer wiedergibt (nicht zu verwechseln mit der Schichtnummerierung):

$$\Delta w_{kj} = \frac{1}{R} \cdot \sum_{r=1}^{R}(t_j^r - o_j^r) \cdot f'_{akt}(\sum_{i=1}^{n} o_i^r \cdot w_{ij}^r) \cdot o_k^r \tag{17}$$

Abschließend muss noch eine sogenannte **Lernrate** η als Koeffizient mit $\eta > 0$ eingeführt werden. Diese ist notwendig zum Skalieren von zuvor erwähnten Schritten, da der

[22]Quelle: `https://en.wikipedia.org/wiki/Logistic_function#Derivative`(besucht: 05.11.2018)
[23]Quelle: *Tangens hyperbolicus und Kotangens hyperbolicus* 2018



(a) Oszillation in steilen Gebieten bei zu großer Lernrate

(b) Langsamer Fortschritt in flachen Gebieten bei zu kleiner Lernrate

Abbildung 9: Mögliche Probleme mit der Wahl der Lernrate (eigene Darstellungen)

Fehler sich durch zu große Änderungen vom Netz über das Minimum hinweg bewegen kann. Im worst case kann es dadurch zu einer endlosen Oszillation um das Minimum kommen. Gleichzeitig verlängert sich aber mit kleiner werdenden Schritten die benötigte Laufzeit zur Approximation eines Minimums, was insbesondere in flach fallenden Tälern bemerkbar wird, siehe dazu Abbildung 9. In der Praxis ist es deswegen sinnvoll, mit einem großen η zu starten und dieses allmählich zu verkleinern, damit erst eine grobe Annäherung mit anschließender Feinoptimierung erfolgen kann. Dies ist auch als **adaptive Lernrate** bekannt (vgl. Ertel, 2013, S.272).

Zusammengefasst ergibt sich für die Berechnung der neuen Gewichte die Formel:

$$w_{neu} \overset{(10)}{=} w_{alt} + \Delta w_{kj} \overset{(17)}{=} w_{alt} + \frac{\eta}{R} \cdot \sum_{r=1}^{R} (t_j^r - o_j^r) \cdot f'_{akt}(\sum_{i=1}^{n} o_i^r \cdot w_{ij}^r) \cdot o_k^r \tag{18}$$

Hierbei ist anzumerken, dass sich Gleichung (18) für die lineare Aktivierungsfunktion mit Steigung $m = 1$ gemäß Tabelle 1 vereinfacht zu

$$w_{neu} = w_{alt} + \Delta w_{kj} = w_{alt} + \frac{\eta}{R} \cdot \sum_{r=1}^{R} (t_j^r - o_j^r) \cdot o_k^r \tag{19}$$

und sie deswegen in der Praxis durchaus häufig für die Delta-Regel angewendet wird. (vgl. ebd.)

3.2 Backpropagation

Wie bereits angesprochen lässt sich die Delta-Regel nicht für mehrschichtige Perzeptren anwenden. Die Lösung für dieses Problem liefert der **Backpropagation-Algorithmus**, zu deutsch etwa „Rückübertragungsalgorithmus“. Dieser ist eine Erweiterung bzw. Verallgemeinerung der Deltaregel und somit ebenfalls ein überwachtes Lernverfahren. Ferner kann er auch sinnvollen Gebrauch vom Bias machen.

3.2.1 Formelherleitung

Das Prinzip des Algorithmus besteht darin, von der Outputschicht aus schichtweise die Änderungen von Gewichten und Bias in Abhängigkeit ihres Einflusses auf die Kostenfunktion zu berechnen und nach einer festgelegten Anzahl an Lernmustern anzuwenden. Der Ansatz lautet dementsprechend für jede Gewichtung $w_{kj}^{(a)}$ einer Schicht a (vgl. Schmiedl, 2006, S. 12):

$$\Delta w_{kj}^{(a)} = -\frac{\partial E_{ges}}{\partial w_{kj}^{(a)}} = -\frac{\partial E_{ges}}{\partial net_j^{(a+1)}} \frac{\partial net_j^{(a+1)}}{\partial w_{kj}^{(a)}} = \delta_j^{(a+1)} \cdot o_k^{(a)} \tag{20}$$

$$\text{mit } \textbf{Fehlersignal } \delta_j^{(a)} := -\frac{\partial E_{ges}}{\partial net_j^{(a)}} \tag{21}$$

Die Ableitung des zweiten Differentials in Gleichung (20) funktioniert hierbei analog zu Gleichung (16). Außerdem soll weiter gelten (vgl. a.a.O. S. 13):

$$\delta_j^{(a)} = -\frac{\partial E_{ges}}{\partial net_j^{(a)}} = -\frac{\partial E_{ges}}{\partial o_j^{(a)}} \frac{\partial o_j^{(a)}}{\partial net_j^{(a)}} = -\frac{\partial E_{ges}}{\partial o_j^{(a)}} \cdot f'_{akt}(net_j^{(a)}) \tag{22}$$

Die Umformungen basieren auf $o_j \overset{(2)}{=} f_{akt}(net_j)$ und der Kettenregel.

Der Sinn hinter dem Fehlersignal ist folgender: Für die Neuronen der verdeckten Schichten stellt sich die Schwierigkeit ein, dass sich ihr direkter Einfluss nur auf die Netzeingabe der Neuronen der jeweils folgenden Schicht beschränkt. Um daraus den Einfluss auf die Kostenfunktion herzuleiten, muss für diese nachfolgenden Neuronen ebenfalls deren Einfluss auf die nachfolgende Schicht berechnet werden, bis es sich um die Ausgabeschicht handelt. In anderen Worten bestimmt man also in Abhängigkeit des Einflusses einer Schicht auf die Kostenfunktion den Einfluss der Schicht davor, wodurch sich dann die Änderungen berechnen lassen.[24] Diese (rekursive) Abhängigkeit kann mit dem Fehlersignal hergeleitet werden, da es genau diesen Einfluss ausdrückt. Die Herleitung der Rekursion geschieht wie folgt:

[24] Daher kommt der Name *Back*propagation.

Die Abbruchbedingung ist das Erreichen der Ausgabeschicht z. Ab dem Moment lässt sich der direkte Einfluss von Gewichtung und Bias auf die Kostenfunktion ermitteln, das Differential $\frac{\partial E_{ges}}{\partial o_j^{(a)}}$ aus Gleichung (22) lässt sich ohne Umwege berechnen (vgl. ebd.):

$$\begin{aligned}
\delta_j^{(z)} = & \quad -\frac{\partial E_{ges}}{\partial o_j^{(z)}} \cdot f'_{akt}(net_j^{(z)}) \\
\overset{(12),(11)}{=} & -\left(\frac{\partial}{\partial o_j^{(z)}} \frac{1}{2} \sum_{i=1}^{n^{(z)}} (t_i - o_i^{(z)})^2 \right) \cdot f'_{akt}(net_j^{(z)}) \\
= & \quad (t_j - o_j^{(z)}) \cdot f'_{akt}(net_j^{(z)})
\end{aligned} \tag{23}$$

Die letzte Umformung funktioniert aufgrund der Tatsache, dass in der Summe nur der Summand $t_j - o_j^{(z)}$ auch $o_j^{(z)}$ enthält, weshalb die anderen Summanden als Konstanten behandelt werden.

Der Rekursionsschritt soll die Abhängigkeit von Neuronen $j^{(m)}$ einer jeden verdeckten Schicht m zur nachfolgenden Schicht $m+1$ darstellen.

Hier lässt sich der Ansatz nicht direkt berechnen, da o_j keinen direkten Einfluss auf die Kostenfunktion hat, stattdessen muss ein weiteres Mal die Kettenregel angewendet und aufgelöst werden (vgl. ebd.):

$$\begin{aligned}
\delta_j^{(m)} &= -\sum_{p=1}^{n^{(m+1)}} \frac{\partial E_{ges}}{\partial net_p^{(m+1)}} \frac{\partial net_p^{(m+1)}}{\partial o_j^{(m)}} \cdot f'_{akt}(net_j^{(m)}) \\
&\overset{(21),(2)}{=} \sum_{p=1}^{n^{(m+1)}} \left(\delta_p^{(m+1)} \frac{\partial}{\partial o_j^{(m)}} b_p^{(m+1)} + \sum_{i=1}^{n^{(m)}} o_i^{(m)} \cdot w_{ip}^{(m)} \right) \cdot f'_{akt}(net_j^{(m)}) \\
&= f'_{akt}(net_j^{(m)}) \cdot \sum_{p=1}^{n^{(m+1)}} \delta_p^{(m+1)} \cdot w_{jp}^{(m)}
\end{aligned} \tag{24}$$

Die Nutzung der Summe in der ersten Zeile basiert darauf, dass jedes Neuron der folgenden Schicht $m+1$ grundsätzlich wenigstens einen indirekten Einfluss auf die Kostenfunktion ausübt, weshalb in dem Fall auch nach jeder Netzmaske dieser Neuronen abgeleitet werden muss. Die Differenzierung der zweiten Summe in der zweiten Zeile geschieht analog zu Gleichung (20).

Zusammengefasst ergeben sich für alle $\delta_j^{(a)}$ mit $1 < a \leq z$ und Outputschicht z folgende Formeln (vgl. ebd.):

$$\delta_j^{(a)} = \begin{cases} f'_{akt}(net_j^{(a)}) \cdot (t_j - o_j^{(a)}) & \text{für } a = z \\ f'_{akt}(net_j^{(a)}) \cdot \sum_{p=1}^{n^{(a+1)}} \delta_p^{(a+1)} \cdot w_{jp}^{(a)} & \text{für } a \neq z \end{cases} \tag{25}$$

Den Bias kann man sich als ein Neuron vorstellen, welches vollständig aktiviert ist, und deren Gewichtung dadurch dem Bias selbst entspricht. Durch die ständige Aktivierung gilt also $o_k^{(a-1)} = 1$, wodurch sich die Gleichung für den Bias direkt aus Gleichung (21) herleitet (vgl. Sanderson, 2017, Teil 4):

$$\Delta b_j^{(a)} = \delta_j^{(a)} \tag{26}$$

3.2.2 Verallgemeinerung

Der letzte Schritt besteht darin, die Formeln an das Batch-Learning (vgl. Abschnitt 3.1.2) anzupassen und mit Hilfe der in Abschnitt 2.2.2 definierten linearen Algebra zu verallgemeinern.

Die Aktualisierung der Gewichte erfolgt analog zu Gleichung (10) mit

$$w_{kj/neu}^{(a)} = w_{kj/alt}^{(a)} + \Delta w_{kj}^{(a)} \tag{27}$$

und kann direkt in Form von Matrizen dargestellt werden:

$$\mathbf{W}_{neu}^{(a)} = \mathbf{W}_{alt}^{(a)} + \Delta \mathbf{W}^{(a)} \tag{28}$$

Für die Fehlersignale lässt sich aus $\sum_{p=1}^{n^{(a+1)}} \delta_p^{(a+1)} \cdot w_{jp}^{(a)}$ in der verallgemeinerten Form ein Matrix-Vektorprodukt bilden, es folgt:

$$\delta^{(a)} = \begin{cases} f'_{akt}(net^{(a)}) \cdot (t - o^{(a)}) & \text{für } a = z \\ f'_{akt}(net^{(a)}) \cdot (\mathbf{W}^{(a)} \cdot \delta^{(a+1)}) & \text{für } a \neq z \end{cases} \tag{29}$$

Ferner entspricht $\delta^{(a+1)} \cdot o^{(a)}$ verallgemeinert dem **dyadischen Produkt**[25], sodass sich für Bias und Gewichtung unter Ergänzung der noch fehlenden Bestandteile des Batch-Learnings insgesamt ergibt:

$$\Delta \mathbf{W}^{(a-1)} = \frac{\eta}{R} \cdot \sum_{r=1}^{R} \delta^{(a),r} \otimes o^{(a-1),r} \tag{30}$$

$$\Delta b^{(a)} = \frac{\eta}{R} \cdot \sum_{r=1}^{R} \delta^{(a),r} \tag{31}$$

(eigene Herleitungen)

[25] Das dyadische Produkt $\otimes$ zweier Vektoren ergibt eine Matrix. Es ist sehr ähnlich zum kartesischen Produkt der Mengenlehre, mit dem Unterschied, dass die Elemente durch normale Multiplikation erhalten werden.

4 Praktische Umsetzung: Ziffern- und Schrifterkennung

Nachdem bisher alle notwendigen Grundlagen erarbeitet wurden, soll im abschließenden Teil der Arbeit ein eigenes künstliches neuronales Netz programmiert werden. Dabei soll es sich um ein mehrschichtiges Perzeptron handeln, welches in einer ersten Variante die zehn Ziffern und in einer zweiten Variante die 26 Buchstaben des normalen Alphabets unabhängig von Groß- und Kleinschreibung erkennen soll.[26] Dies soll mit Python 3 geschehen. Zur Verdeutlichung der Inhalte dieser Arbeit habe ich mich ferner gegen das Verwenden von speziellen Bibliotheken entschieden, die bereits Funktionen zur Erzeugung von neuronalen Netzen zur Verfügung stellen.[27]

4.1 Dateneinspeisung

Zunächst ist es notwendig, dass die Inputdaten nicht nur in einer verwertbaren Form, sondern ebenfalls in genügend großen Menge vorliegen.

4.1.1 Bereitstellung von Trainingsdaten

Allgemein gilt: Je mehr Vielfalt in den Trainingsdaten, desto höher auch die Genauigkeit des Netzes. Aus diesem Grund reicht es nicht, selbst jeden Buchstaben einige Male zu zeichnen und zu digitalisieren. Stattdessen ist es effizienter, sich einer Datensammlung mit wenigstens einigen zehn- bis hunderttausend Trainingsdaten zu bedienen, die von einer Vielfalt an Personen mit verschiedenen Handschriften erstellt wurden. Da das neuronale Netz eine feste Anzahl an Neuronen hat, ist es außerdem notwendig, dass über jedes Zeichen jeweils die gleiche Menge an Information gespeichert ist, die eingespeist werden soll. Weil es sich jedoch um Bilder handelt, Neuronen aber nur mit Zahlen arbeiten, müssen die Bilddaten erst auf reine Zahlenwerte heruntergebrochen werden können. Dazu genügt es, den Farbwert eines jeden Pixels an einer bestimmten Position einem Inputneuron zuordnen zu können.

Diese Voraussetzungen erfüllt die **MNIST-Datensammlung** für Ziffern sowie die **EMNIST-Datensammlung** für Buchstaben.[28] Neben einem Trainingsset besitzen diese bereits ein separates Set zur Überprüfung des Grades der Überanpassung.

Zu beachten ist das Speicherformat der einzelnen Sets: Diese bestehen jeweils aus einer Datei, welche die eigentlichen Trainingsmuster mit den Bilddimensionen 28x28 Pixel enthält, und aus einer Datei, die in der selben Reihenfolge die Klasse (hier: *Label*)

[26] Wie sich durch Ausprobieren festgestellt hat, würde eine Unterteilung in Groß- und Kleinbuchstaben zu viel Rechenzeit benötigen und zu ungenau in der Erkennung werden.

[27] Ein beliebtes Beispiel dafür wäre `TensorFlow`.

[28] Download verfügbar unter `http://yann.lecun.com/exdb/mnist/` (oben) bzw. `https://www.nist.gov/itl/iad/image-group/emnist-dataset` (ganz unten, als *binary format* (Stand: 04.11.2018)

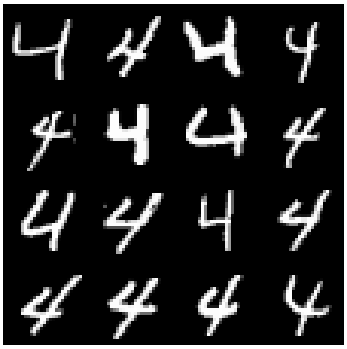
	Trainingsset	Überprüfungsset	Gesamt
`MNIST`	60.000	10.000	70.000
`EMNIST-Letter`	124.800	20.800	145.600

Tabelle 2: Anzahl der Trainingsmuster pro Set
(jedes Zeichen kommt in seinem Set identisch oft vor)

(a) `EMNIST`: ungeordnet

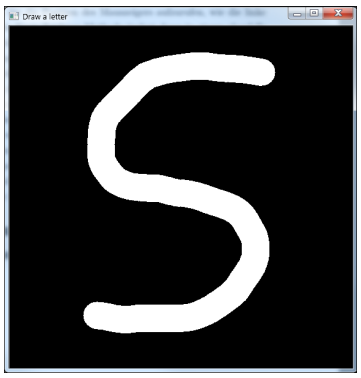
(b) `EMNIST`: Buchstabe P

(c) `MNIST`: ungeordnet

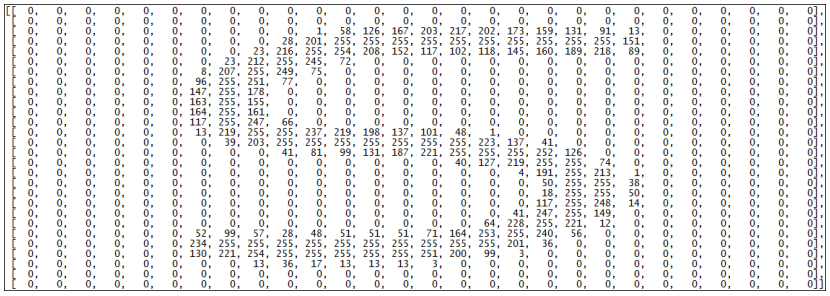


(d) `MNIST`: Ziffer 4

Abbildung 10: Ausschnitte aus der `MNIST`- und `EMNIST-Letter`-Datenmenge
(eigene Darstellungen; siehe Anhang A.3)



(a) 560x560-Zeichenfenster mit Buchstabe S



(b) komprimiertes 28x28-`numpy`-Feld des gleichen Buchstabens

Abbildung 11: Arbeitsweise des Programms zur eigenen Erstellung von Buchstaben
(eigene Darstellungen)

der Trainingsmuster beinhaltet.[29] Besonders ist, dass es sich bereits um *Binärdateien* handelt. Für die Trainingsmuster bedeutet das, dass jeweils $(28 \cdot 28 =)784$ aufeinanderfolgende Bytes ein Bild ergeben und dabei jedes Byte den Grauwert eines Pixels als einen `integer`-Wert von 0 (schwarz) bis 255 (weiß) repräsentiert. Damit entfällt das Problem, dass Bilder erst in ihre Pixelwerte umgewandelt werden müssen, was Zeit und Aufwand spart. Ähnliches gilt für die Label, bei denen jedes Byte (außer am Anfang) für ein Zeichen steht. (vgl. LeCun, Cortes und Burges, 2013)[30] Dadurch lassen sich die Bytewerte direkt als `integer` in je zwei `numpy`-Felder pro Set initialisieren. `numpy` ist eine Bibliothek, die hocheffiziente Berechnungen mit Feldern (was quasi Vektoren und Matrizen entspricht) zur Verfügung stellt, mehr dazu siehe Abschnitt 4.2.2. Für die genaue Implementierung siehe Anhang A.2. Eine Übersicht der Trainingsmuster ist in Tabelle 2 zu sehen. Ausschnitte der Zahlen und Buchstaben sind in Abbildung 10 zu erkennen. (vgl. LeCun, Cortes und Burges, 2013; Cohen u. a., 2017)

4.1.2 Einspeisen von eigenen Inputdaten

Nach dem Trainieren des Netzes ist es auch nützlich, eigene handschriftliche Buchstaben bestimmen zu lassen. Hierzu kann man sich der `OpenCV`-Bibliothek[31] bedienen, die Echtzeit-Methoden zur Bildbearbeitung anbietet und Bilddaten direkt in einem `numpy`-Feld abspeichert. In unserem Fall lässt sich eine Methode definieren, die aufgerufen wird, sobald es ein Mausevent (wie Bewegen, Klicken, etc.) in einem zuvor definierten und erzeugten Zeichenfenster gab. Je nach exaktem Event lassen sich dann verschiedene Reaktionen implementieren. Hier genügt es, die mitgelieferte Kreis-Zeichnen-Methode solange für die aktuelle Position des Mauszeigers aufzurufen, wie die linke Maustaste gedrückt ist. Die Kreis-Zeichnen-Methode ändert dann in einem ebenfalls zuvor definierten `numpy`-Feld entsprechende Werte, die im Anschluss im Zeichenfenster angezeigt werden. Wenn nun eine Ziffer bzw. ein Buchstabe gezeichnet wurde, lässt sich das Fenster schließen und das Feld direkt weiterverwenden.

Wenn das Feld jedoch genauso groß wie die Auflösung der Trainingsmuster ist - nämlich 28x28 Pixeln - dann ist das daraus resultierende Fenster deutlich zu klein. Aus diesem Grund muss man ein größeres Fenster mit einem größeren Feld erzeugen, welches dafür im Nachhinein herunterskaliert werden muss. Dies ist möglich, indem man den Durchschnittswert einer bemalten, rechteckigen Fläche berechnet und als einzelnes Ele-

[29] Da die Dateien anfangs noch im `.gz`-Format komprimiert sind, müssen diese vor dem ersten Gebrauch noch entpackt werden. Für Windows ist hier das kostenfrei im Internet verfügbare Programm **7zip** empfehlenswert, da das systeminterne Entpackungsprogramm diese Komprimierung nicht kennt.

[30] In den Bytedateien sind anfangs noch die Eigenschaften der Muster gespeichert, für das Verständnis des allgemeinen Prinzips ist dies jedoch nicht von Belang. In der Quelle gibt es dazu genaue Informationen.

[31] `OpenCV` muss über den Python-Updater zunächst manuell installiert werden.

ment in einem neuen Feld der Größe 28x28 abspeichert. So kann man beispielsweise ein Fenster der Größe 560x560 Pixel erzeugen und jeweils 20x20-Rechtecke auf ein Element herunterbrechen, siehe Abbildung 11 für eine Veranschaulichung. Der Programmcode hierfür ist unter Anhang A.4 einsehbar.

4.2 Programmierung des künstlichen neuronalen Netzes

4.2.1 Funktionen des Netzes

Das künstliche neuronale Netz soll eine eigene Klasse sein, deren Instanzen sowohl trainiert als auch getestet werden sollen. Viele Teile des Codes lassen sich hierbei direkt aus den bereits erarbeiteten Formeln aus Abschnitt 3.2.2 mithilfe der `numpy`-Bibliothek implementieren. Wie bereits angesprochen lassen sich die Felder, die `numpy` zur Verfügung stellt, als Vektoren und Matrizen utilitarisieren. Der große Vorteil von `numpy` ist nun, dass es nicht nur bereits alle möglichen Methoden zum Rechnen mit Vektoren mitliefert, sondern dabei auch hocheffizient ist. Einer der Gründe dafür ist die Tatsache, dass es durch **Multithreading** alle CPU-Kerne gleichzeitig zum Berechnen nutzt und dadurch Rechenzeiten drastisch verkürzt. Dies hängt wiederum damit zusammen, dass normale Python-Programme stets nur einen Kern nutzen und Multithreading nur mit hohem, zusätzlichem Programmieraufwand implementiert werden kann, was den Rahmen dieser Arbeit sprengen würde.[32] Zu beachten ist, dass das Netz eine unterschiedliche Anzahl an verdeckten Schichten haben können soll. Darum bestehen viele Teile des Codes daraus, dass jeder Wert für jede Schicht in einer (Standard-Python-)Liste abgespeichert werden soll, deren Länge von der Schichtanzahl abhängt. Ansonsten basiert der Code darauf, dass er während der Lernphase eine anfangs festgelegte Anzahl an Epochen jeweils einen Minibatch aus den Trainingsdaten zusammenstellt und die durchschnittliche Änderung der Gewichtungen und Bias berechnet. Dies geschieht, indem für jedes Trainingsmuster im Minibatch der Netzoutput (Feedforward) und darauf basierend den Gesamtfehler zu minimieren versucht (Backpropagation). Zum Klassifizieren von unbekannten Werten genügt es, nach Übergabe des Inputs per Feedforward den Netzoutput zu berechnen, und von diesem wiederum den größten Wert zu nehmen. Die Position des Outputneurons entspricht dann der Klasse. Der Code mitsamt Anmerkungen ist in Anhang A.5 einsehbar.

4.2.2 Benutzung des Netzes

Folgende Eigenschaften werden durch die Instanzierung festgelegt (beachte hierzu auch den Anfang von Anhang A.5):

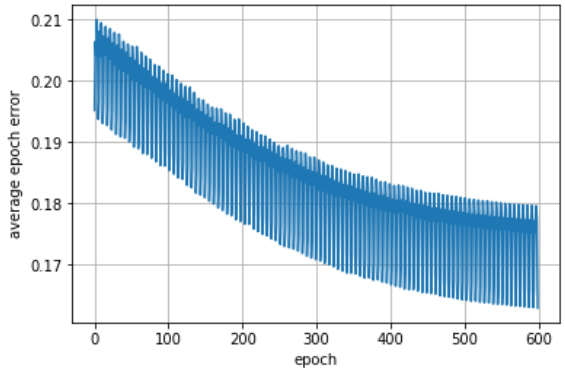
[32] Heutige Rechner besitzen oft 4 bis 8 Kerne, woraus etwa eine 4- bis 8-fache Leistungssteigerung resultiert.

- Netzstruktur: Sie wird in Form eines `integer`-Tupels, welches nacheinander die Neuronen jeder Schicht beginnend beim Input wiedergibt, festgelegt.
- Aktivierungsfunktionen: Sie werden in Form eines `String`-Tupels, welches nacheinander für jede Schicht die Aktivierungsfunktion wiedergibt, festgelegt.
- sowie Epochenanzahl, adaptive Lernrate, Minibatchgröße, Mischen der Muster, Seed und Zwischenspeicherungen des Netzes

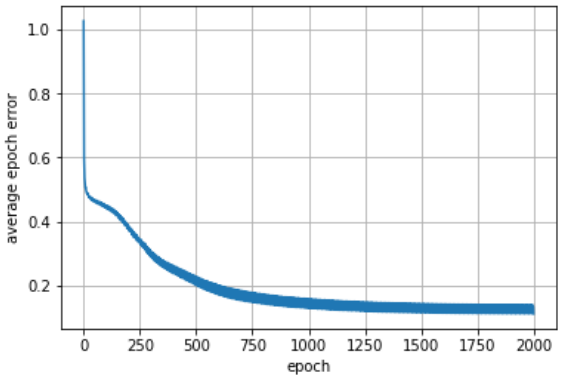
Um ein Netz in die Lernphase zu versetzen, müssen zunächst die Trainingsmuster und -label in die `numpy`-Felder geladen werden. Außerdem können vorher noch die Speicherorte von Gewichtungen und Bias in zwei Listen gespeichert werden, um statt neuen Werten diese alten zu benutzen. Dies ist nützlich, wenn nach einer Lernphase noch eine weitere erfolgen soll, die an jener anschließt. Der Aufruf der Lernmethode geschieht dann durch Mitgabe der entsprechenden Werte. Im Anschluss können verschiedene Werte (insbesondere der Verlauf der Gesamtfehler) mithilfe der `Matplotlib`-Bibliothek in Form von Graphen veranschaulicht werden. Außerdem kann das Netz entweder durch entsprechendes Laden der Sets oder durch das eigene Zeichnen getestet werden. Ein Codebeispiel mit Anmerkungen ist im Anhang A.6 zu sehen.

4.3 Analyse der Testergebnisse

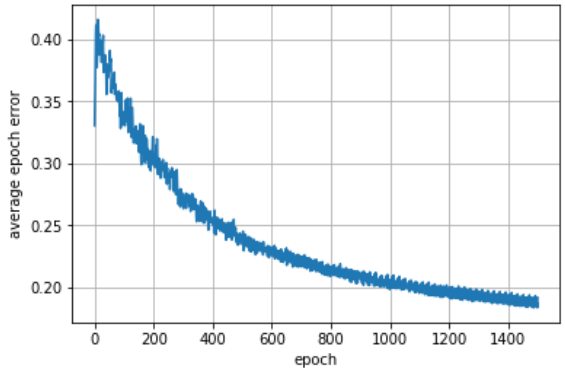
Nach einigen Tests (siehe Anhang B) konnten folgende eigene Beobachtungen gemacht werden:
Die markanteste Feststellung ist der Laufzeitunterschied zwischen der Ziffern- und der Buchstabenerkennung, welcher sich um einen Faktor 10 bis 20 unterscheidet. Dies war zu erwarten, da die Buchstabenerkennung aufgrund der höheren Vielfalt der Outputmöglickeiten zwingend mehr Neuronen braucht, mit denen wiederum die Vektor- und Matrizengröße steigt. Zudem lohnt sich ein Blick auf den Verlauf des durchschnittlichen Gesamtfehlers jeder Epoche: Wie in Abbildung 12(a) beispielsweise gut zu erkennen ist, schwankt der durchschnittliche Fehler sehr stark. Tatsächlich ist diese Schwankung periodisch. Sie lässt sich dadurch erklären, dass die Minibatches, die sich untereinander alle unterscheiden, immer eine leicht andere Landschaft an Bergen und Tälern erzeugen. Deswegen ist es auch notwendig, stets eine hinreichend große Minibatchgröße zu benutzen, da andernfalls die Schwankungen zu stark werden und das Netz nicht konvergieren kann. Interessant ist zudem Abbildung 12(b), in der das Netz sich bis zur ca. 150. Epoche relativ langsam verbessert, nahezu direkt danach aber deutlich schneller ist. Dies könnte einerseits daran liegen, dass das Netz ein flaches Plateau überwunden hat und nun an einem „Hang“ ist. Andererseits ist es auch möglich, dass die Lernrate zuvor zu hoch war und das Netz zu sehr pro Epoche gesprungen ist.



(a) Netz I Phase 2.



(b) Netz II Phase 1.



(c) Netz IV Phase 4.

Abbildung 12: Durchschnittlicher Gesamtfehler verschiedener Lernphasen (eigene Darstellungen; siehe Anhang A.6)

Bei einem Vergleich zwischen dem zweischichtigen Netz (II)[33] und dem dreischichtigen Netz (III) zur Ziffernerkennung lässt sich feststellen, dass es in diesem Fall weder nennenswerte Vor- noch Nachteile gab.

Besonders beim Netz für die Schrifterkennung (IV) war, dass dieses eine deutlich höhere Lernrate benötigt hat als die Ziffernerkennung. Aus diesem Grund hat das Netz erst in der vierten Phase große Fortschritte gemacht, als die anfängliche Lernrate 20 betrug und somit 25-mal so hoch wie bei der Ziffernerkennung war, siehe hierzu auch Abbildung 12(c).

Betrachtet man die Verteilung, wie oft welches Zeichen falsch gedeutet wurde, so fällt auf, dass manche besonders häufig falsch bzw. richtig bestimmt werden. Insbesondere scheint sich diese Verteilung unabhängig von der Netzstruktur zu erhalten. Beispielsweise sind die Ziffern 5 und 8 die Ziffern, die in den seltensten Fällen erkannt wurden (X: 92%), während 1 und 6 am häufigsten (X: 97%) erkannt wurden. Bei den Buchstaben wurde kein einziges „e“ und kein einziges „s“ als solches gedeutet, während „m“ und „o“ mit 90% am besten bestimmt wurden. Ein Teil der Erklärung könnte darin liegen, dass sich manche Ziffern zu ähnlich sehen, als dass das Netz sie (ohne viel Lernen) eindeutig voneinander unterschieden kann. Außerdem haben bestimmte Zeichen teils sehr indivuelle Schreibweisen pro Verfasser. Dies erschwert es dem Netz, die Muster vollständig zu lernen.

Allgemein ist die Leistung der Buchstabenerkennung noch nicht wirklich gut, dafür ist das Netz im Gegensatz zum Ziffernnetz X noch nicht konvergiert (was aber noch mehr Zeit in Anspruch nehmen würde).

Abschließend lässt sich für das Testen mit eigenen Buchstaben noch hinzufügen, dass sie stets mittig und platzfüllend eingetragen werden müssen (wie es die Trainingsdaten auch sind). Andernfalls trifft das Netz fast immer falsche Entscheidungen.

[33] Nummerierung basierend auf Anhang B

5 Fazit

In dieser Arbeit wurde Schritt für Schritt dargelegt, wie einfache künstliche neuronale Netze funktionieren, und wie sich daraus ein Programm schreiben lässt. Man konnte sehen, wie mithilfe von Analysis einige „schöne“ Formeln hergeleitet werden können, mit denen etwas so scheinbar Komplexes wie Mustererkennung vom Konzept her relativ einfach wird. Insbesondere ist bemerkenswert, dass das sogenannte „Lernen“ des Netzes letzten Endes auf reiner Mathematik aufbaut. Es wurde deutlich, dass einfache Netze zwar bereits sinnvoll angewendet werden können, diese aber noch nicht effizient genug sind, wie sich beim Abschneiden bei der Buchstabenerkennung bemerkbar gemacht hat. Außerdem ersichtlich wurde der Faktor der Rechenleistung: Selbst für die Schrifterkennung hat ein herkömmlicher PC viele Stunden gebraucht. Daraus lässt sich folgern, warum neuronale Netze erst in der heutigen Zeit zu hohem Ansehen gekommen sind, da früher bei weitem noch nicht die Kapazitäten existiert hatten. Das zeigt aber auch die Bedeutsamkeit von anfangs erwähnten Supercomputern für künstliche neuronale Netze. Erst diese haben die nötige Leistung für komplexe Anwendungen, die dann teilweise auch kommerziell eingesetzt werden können, wie es beispielsweise Google macht, um anhand von Nutzerverhalten zugeschnittene Empfehlungen anbieten zu können (vgl. Covington, Adams und Sargin, 2016).

Zwar bleibt noch offen, warum es dem Menschen im Gegensatz zu Computern teils immer noch derart leicht fallen kann, seine Umgebung so umfassend zu erkennen. Trotzdem wird es in Anbetracht der kontinuierlichen Fortschritte mit großer Wahrscheinlichkeit einen Zeitpunkt geben, an denen der Computer den Menschen diesbezüglich übertrumphen wird. Und es ist nicht unwahrscheinlich, dass künstliche neuronale Netze einen großen Beitrag dazu leisten werden. Insbesondere würde dadurch die Konstruktion eines autarken Systems, beispielsweise in Form eines Roboters, stark begünstigt werden. Anfangs mag dies etliche Vorteile mit sich bringen: Viel Hausarbeit könnte automatisiert werden, Autos fahren völlig unabhängig zu ihren Zielen, und so weiter. Doch wie bei ziemlich vielen (wenn nicht sogar allen) technischen Entwicklungen auch sind damit stets Gefahren verbunden: Genauso wie bei der Kernspaltung nicht bloß der Aspekt der billigen Alternative zur Stromerzeugung in den Vordergrund gerückt werden sollte, sondern auch damit verbundene Probleme wie Atommüll, Sicherheit und zukünftige Verfügbarkeit, genauso sollten fortgeschrittene Roboter nicht bloß als Maschinen gesehen werden, die das Leben des Menschen - ganz banal gesagt - versüßen: Es sollte augenscheinlich sein, dass solche Erfindungen aufgrund ihres hohen Leistungspotenzials dann auch vermehrt Einsatz in der kapitalistisch-wettbewerbsorientierten Industrie haben werden, und somit immer mehr Berufe obsolet machen würden. Zwar existieren bereits jetzt etliche Rationalisierungsprozesse, doch solche Entwicklungen

würden nicht nur manche, sondern wahrscheinlich alle Berufe betreffen, die eine erlernbare Tätigkeit zu Grunde liegen haben. Vor allem wenn Roboter auch die Emotionalität eines Menschen erlernen und nachahmen könnten, würde das heißen, dass es bestenfalls nur noch schöpferisch-intellektuelle Berufe wie Forschung geben könnte. So oder so würde es vielen in unserer heutigen Gesellschaftsstruktur die Existenzgrundlage entreißen. Man mag vielleicht entgegenstellen, dass man doch stets umlernen kann, doch einerseits würde es wahrscheinlich nicht mehr genug Berufsmöglichkeiten für alle geben, andererseits sind für die verbleibenden Berufe sowohl eine gewisse intellektuelle Begabung als auch Motivation essentiell, die gewiss nicht jeder innehat. Unter dem Strich bedeutet das, wenn es in so einem Szenario keine strukturellen Umbrüche gäbe, würde es zur Massenverelendung kommen.

Den Fortschritt kann man nur schwer aufhalten, trotzdem könnten solche Dystopien vermieden werden: Ich finde, es sollte eine fundamentale Rolle in der Tätigkeit des Wissenschaftlers spielen, insbesondere bei großen Entwicklungen stets in Kontakt mit Vertretern der Gesellschaft (wie Politikern) zu stehen, um etliche Folgen abzuwägen sowie präventiv Gegenmaßnahmen zu etwaigen Eventualitäten zu entwickeln. Mit diesen Worten möchte ich den Leser auch selbst dazu anregen, sich einmal selber über die Konsequenzen von technischen Entwicklungen - ob bereits geschehen, oder zukünftig - bewusstzuwerden, denn es betrifft immer uns alle, und wird uns immer betreffen!

6 Literatur

Aunkofer, Benjamin (2. Juli 2017). *Überwachtes vs unüberwachtes maschinelles Lernen*. URL: https://data-science-blog.com/blog/2017/07/02/uberwachtes-vs-unuberwachtes-maschinelles-lernen/ (besucht am 05. 11. 2018).

Cohen, G. u. a. (1. März 2017). *EMNIST: an extension of MNIST to handwritten letters*. Wissenschaftliche Arbeit. URL: https://arxiv.org/pdf/1702.05373v2.pdf (besucht am 05. 11. 2018).

Covington, Paul, Jay Adams und Emre Sargin (Sep. 2016). *Deep Neural Networks for YouTube Recommendations*. Wissenschaftliche Arbeit. URL: http://static.googleusercontent.com/media/research.google.com/de//pubs/archive/45530.pdf (besucht am 05. 11. 2018).

Ertel, Wolfgang (2013). *Grundkurs Künstliche Intelligenz*. 3. Auflage. Wiesbaden: Springer Vieweg Verlag.

Gableske, Oliver (16. Apr. 2005). *Multilayer-Perzeptron*. Seminarskript. URL: http://www.informatik.uni-ulm.de/ni/Lehre/WS04/ProSemNN/pdf/MLP.pdf (besucht am 05. 11. 2018).

Görz, Günther, Josef Schneeberger und Ute Schmid (2014). *Handbuch der Künstlichen Intelligenz*. 5. Auflage. München: Oldenbourg Verlag.

Hyperebene (Nov. 2018). URL: https://de.wikipedia.org/wiki/Hyperebene (besucht am 05. 11. 2018).

Kroll, Andreas (2013). *Computational Intelligence*. Oldenbourg Verlag.

Kruse, Rudolf (15. Juli 2011). *Neuronale Netze*. Vorlesungsskript. URL: http://fuzzy.cs.ovgu.de/ci/nn/nn-all.pdf (besucht am 05. 11. 2018).

Lämmel, Uwe und Jürgen Cleve (2008). *Künstliche Intelligenz*. 3. Auflage. München: Hanser Verlag.

LeCun, Yann, Corinna Cortes und Christopher Burges (2013). *THE MNIST DATABASE of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/ (besucht am 05. 11. 2018).

Mouse as a Paint-Brush (10. Nov. 2014). URL: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_gui/py_mouse_handling/py_mouse_handling.html (besucht am 05. 11. 2018).

mplot3d tutorial (10. Mai 2017). URL: https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#surface-plots (besucht am 05. 11. 2018).

Raschka, Sebastian (2017). *Machine Learning mit Python*. Frechen: mitp Verlags GmbH & Co. KG.

Sanderson, Grant (Okt. 2017). *Neural networks*. URL: https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi (besucht am 05. 11. 2018).

Schmiedl, Christina (27. Mai 2006). *Neuronale Netze: mehrschichtige Perzeptrone*. Seminarskript. URL: http://www.cogsys.cs.uni-tuebingen.de/lehre/ss06/pro_learning/MLPSchmiedl.pdf (besucht am 05. 11. 2018).

Sunway TaihuLight (Mai 2018). URL: https://en.wikipedia.org/wiki/Sunway_TaihuLight (besucht am 05. 11. 2018).

Supercomputer (Mai 2018). URL: https://de.wikipedia.org/wiki/Supercomputer (besucht am 05. 11. 2018).

Tangens hyperbolicus und Kotangens hyperbolicus (Juni 2018). URL: https://de.wikipedia.org/wiki/Tangens_hyperbolicus_und_Kotangens_hyperbolicus (besucht am 05. 11. 2018).

Zuse Z3 (Mai 2018). URL: https://de.wikipedia.org/wiki/Zuse_Z3 (besucht am 05. 11. 2018).

7 Abbildungsverzeichnis

8 Tabellenverzeichnis

A Programmcode

Um die Mischung von Deutsch und Englisch (durch importierte Methoden) im Code zu vermeiden, ist dieser vollständig in Englisch verfasst. Allein die Kommentare sind in Deutsch.

Benutzte Bibliotheken

```
# Python Image Library (einfache Bildbearbeitung)
from PIL import Image
import numpy as np
# OpenCV (Echtzeit-Bildbearbeitung)
import cv2
# Plotting
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator,
    FormatStrFormatter
# Zeitmessung
import time
# erleichtert das Einlesen von Bytefolgen
import struct
```

A.1 Plotten der Abbildungen

Plotten der Aktivierungsfunktionen; eigene Implementierung:

```
def logisticFunc(net,c = 1):
    return 1 / (1 + np.exp(-c*net))

def linearFunc(net,m = 1):
    return m * net

def tanHypFunc(net,c = 1):
    return 1 - 2 / (np.exp(2*net*c) + 1)

def heavisideFunc(net):
    return np.heaviside(net,np.nan)
```

```
12
13 def partwiselinearFunc(net, theta = 1):
14     return np.piecewise(net, [net < -theta, net > theta],
15                         [0,1,lambda net: 0.5 + net / (2 * theta)])
16
17 values = np.linspace(-11,11,11001)
18
19 f1 = logisticFunc(values,0.5)
20 f2 = logisticFunc(values)
21 f3 = logisticFunc(values,2)
22 # wiederhole für beliebig viele Werte
23
24 plt.figure(1) # gibt den aktuellen Plot an
25 plt.plot(values,f1,lw = 1.5)
26 plt.plot(values,f2,lw = 1.5)
27 plt.plot(values,f3,lw = 1.5)
28 # wiederhole für alle anderen Werte
29
30 plt.xlim(-10, 10)
31 plt.title("Logistische Funktion")
32 plt.ylabel("Output")
33 plt.grid(True)
34 plt.xlabel("Netzeingabe")
35 plt.legend(["c = 0.1","c = 0.2", "c = 0.5","c = 1","c = 2","c
      = 4","c = 100"])
36 plt.savefig("logFunction.png", bbox_inches="tight")
37
38 # wiederhole für beliebig viele vers. Funktionen:
39 g1 = linearFunc(values,0.2)
40 # etc.
41
42 plt.figure(2)
43 plt.plot(values,g1,lw = 1.5)
44 # analog zu oben weiter
45
46 plt.show()
```

Plotten zufälliger Datenmengen zur beispielhaften Veranschaulichung von Klassifizierung; eigene Implementierung:

```
np.random.seed(47801)

N = 80
r0 = 0.5
# Generierung von Punkten mit zufälliger Position
x = 0.9 * np.random.uniform(-1.0,1.0,N)
y = 0.9 * np.random.rand(N)
area = np.full(N,40)
# Zuordnung der Punkte einer Klasse
r = np.sqrt(x * x + y * y)
area1 = np.ma.masked_where(r < r0, area)
area2 = np.ma.masked_where(r >= r0, area)
a = plt.scatter(x, y, s=area1, marker="x", c="b", label = "1.
    Datenset")
b = plt.scatter(x, y, s=area2, marker="o", c="g", label = "2.
    Datenset")
# Anzeige der Grenzlinie:
theta = np.arange(0, np.pi / 2, 0.01)
plt.plot(r0 * np.cos(theta), r0 * np.sin(theta), c="#F01000")
plt.plot(r0 * -np.cos(theta), r0 * np.sin(theta),c="#F01000")
r = mlines.Line2D([], [], color="red", label="Optimale
    Trennlinie")

plt.ylabel("Wert von Eigenschaft 2")
plt.xlabel("Wert von Eigenschaft 1")
plt.yticks(np.arange(0,0, 2))
plt.xticks(np.arange(0,0, 2))
plt.legend(handles = [a,b,r])
plt.savefig("notLinSeparabel.png", bbox_inches="tight")
plt.show()
```

3D-Plotten einer Fehlerfunktion; basierend auf Beispiel von *mplot3d tutorial* 2017, leicht angepasst:

```
fig = plt.figure(figsize=(8, 5))
ax = fig.gca(projection='3d')
ax.set_xlabel("Gewichtung 1")
ax.set_ylabel("Gewichtung 2")
ax.set_zlabel("Wert der Kostenfunktion")
```

```
6
7 # Erstelle Daten
8 X = np.arange(-10, 4, 0.05)
9 Y = np.arange(-10, 4, 0.05)
10 X, Y = np.meshgrid(X, Y)
11 R = np.sqrt(np.exp(X)+ np.exp(Y))
12 Z = np.sin(R) + 1
13
14 # Plotte die Oberfläche
15 surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
16                        linewidth=0, antialiased=True)
17
18 # Individualisierung der Z-Achse
19 ax.set_zlim(0,2)
20 ax.zaxis.set_major_locator(LinearLocator(11))
21 ax.zaxis.set_major_formatter(FormatStrFormatter('%.01f'))
22
23 # Farbskala
24 fig.colorbar(surf, shrink=0.5, aspect=10)
25 plt.savefig("errorFunction.png", bbox_inches="tight")
26
27 plt.show()
```

A.2 Initialisierung der Trainingsmuster

basierend auf Raschka, 2017, S.353, leicht angepasst:

```
1 def setImageArray(img_dataset,lab_dataset):
2     # öffnen der Label-Datei als Bytefolge ("rb")
3     with open(lab_dataset,"rb") as binary_data:
4         # Einlesen der Trainingsmusteranzahl ("num")
5         magic, num = struct.unpack(">II",binary_data.read(8))
6         # Initialisierung eines 1D-numpy-Feldes mit
              Labelwerten als Elemente
7         label_array = np.fromfile(binary_data,dtype=np.uint8)
8     # öffnen der Trainingsmuster-Datei als Bytefolge
9     with open(img_dataset,"rb") as binary_data:
10        # Einlesen der Musteranzahl sowie Bilddimension
```

```
11          magic_num, image_num, rows_num, columns_num = struct.
                unpack(">IIII",binary_data.read(16))
12          # Initialisierung eines 2D-numpy-Feldes mit
                Pixelwerten als Elemente
13          image_array = np.fromfile(binary_data, dtype=np.uint8)
                .reshape(image_num,rows_num*columns_num)
14      if image_num != num:
15          raise ValueError("Amount of images and labels not
                identical!")
16      return image_array, label_array, rows_num, columns_num
```

Hinweis: Die Werte `magic` und `magic_num` haben für das Programm keine Relevanz, da sie aber in den ersten vier Bytes der Dateien abgespeichert sind, müssen sie miteingelesen werden.

A.3 Umwandlung der Bytedaten in Bilder

eigene Implementierung; erzeugt aus allen Mustern ein großes Bild:

```
1  def createInputPictures(dataset,savefile,num_pics_length,
       num_pics_height):
2      # öffnen der Trainingsmuster-Datei als Bytefolge
3      with open(dataset,"rb") as binary_data:
4          # Einlesen der Trainingsmuster-Eigenschaften
5          magic_num, image_num, rows_num, columns_num = struct.
               unpack(">IIII",binary_data.read(16))
6          px_per_img = columns_num * rows_num
7          image_array = np.fromfile(binary_data, dtype=np.uint8)
               .reshape(image_num,rows_num*columns_num)
8          # Größe des Output-Bildes in Musteranzahl
9          length = num_pics_length
10         heigth = num_pics_height
11         if length * heigth > image_num:
12             print("Requested output dimensions ({},{}) needs
                   {} images, but input only provides {}!".format(
                   length,heigth,length*heigth,images_number))
13             print("!!! ABORTED OUTPUT GENERATION !!!")
14             return None
15         print("This dataset includes",image_num,"images @",
               columns_num,"x",rows_num,"pixels.")
```

```
16          go_on = input("Do you want to continue processing {}
                pixels? (y/n)".format(px_per_img*image_num))
17          if go_on != "y":
18              print("!!! ABORTED OUTPUT GENERATION !!!")
19              return None
20          # Generierung eines leeren Bildes mit gewünschten
                Dimensionen
21          picture = Image.new("L", (columns_num*length,rows_num*
                heigth))
22          for h in range(heigth):
23              for l in range(length):
24                  # Daten des aktuellen Buchstabens
25                  letter = image_array[h*length+l].reshape(
                        columns_num,rows_num)
26                  # Achsenvertauschung
27                  letter = letter.T
28                  # Generierung eines temporären Bildes vom
                        aktuellen Buchstaben
29                  tempPic = Image.fromarray(letter)
30                  # Einfügen in das große Bild
31                  picture.paste(tempPic,(l*columns_num,h*
                        rows_num))
32          print("Successfully created output image @",savefile,"
                !")
33          picture.save(savefile)
```

eigene Implementierung; erzeugt für jedes Label ein großes Bild beliebiger Größe:

```
1  def sortedInputPictures(dataset,labelset,max_pics_length,
       max_pics_height):
2      print("Start customized generation of input pictures...")
3      # Initialisierung der Muster- + Labeldaten
4      images, labels, vertical, horizontal = setImageArray(
           dataset, labelset)
5      length = max_pics_length
6      height = max_pics_height
7      # Iteration aller Labelmöglichkeiten
8      for n in range(1,27):
9          # gefundene Bilder mit passendem Label
```

```
10          imgnum = 0
11          # Generierung eines leeren Bildes mit gewünschten
                Dimensionen
12          picture = Image.new("L", (horizontal*length,vertical*
                height))
13          # Durchtesten aller Muster
14          for i in range(len(labels)):
15              if labels[i] == n:
16                  image = images[i].reshape(vertical,horizontal)
17                  # Achsenvertauschung
18                  image = image.T
19                  # Generierung eines Bildes von einem
                        Buchstaben
20                  tempImage = Image.fromarray(image)
21                  # Einfügen in großes Bild
22                  picture.paste(tempImage,((imgnum % length)*
                        horizontal,(int(imgnum / length))*vertical)
                        )
23                  imgnum += 1
24                  # frühzeitiger Abbruch, wenn großes Bild
                        gefüllt
25                  if imgnum == length * height:
26                      break
27          save_string = "char{}-dim_{}x{}.png".format(n,length,
                height)
28          picture.save(save_string)
29      print("Done!")
```

Hinweis: Im Gegensatz zu den `MNIST`-Mustern sind die `EMNIST`-Muster bezüglich der Bildachsen vertauscht. Aus diesem Grund ist die Achsenvertauschung im Code auch nur für `EMNIST` notwendig.

A.4 Erstellung von eigenen Mustern zur Überprüfung in Echtzeit

`compressArray`: eigene Implementierung; Rest basierend auf *Mouse as a Paint-Brush* 2014, angepasst:

```
1 # Methode für Mausevent:
2 # solange linke Maustaste gedrückt, "male" eine Linie
      bestehend aus Kreisen
```

```
3  def drawCircle(event,x,y,flags,param):
4      global drawing
5      if event == cv2.EVENT_LBUTTONDOWN:
6          drawing = True
7      elif event == cv2.EVENT_MOUSEMOVE:
8          if drawing == True:
9              cv2.circle(custom_img,(x,y),param,255,-1)
10     elif event == cv2.EVENT_LBUTTONUP:
11         drawing = False
12         cv2.circle(custom_img,(x,y),param,255,-1)
13
14 def compressArray(array, goal_size):
15     # Dimensionen des ursprünglichen Feldes
16     shape = array.shape
17     # Initialisierung des neuen, komprimierten Feldes
18     compressed_array = np.zeros(goal_size, np.uint8)
19     if not (shape[0] % goal_size[0] == 0 and shape[1] %
           goal_size[1] == 0):
20         print("Cannot compress array of shape {} into shape
               {}! \n Target shape needs to be divisor of original
               shape!").format(shape,goal_size)
21         return compressed_array
22     x_compressionrate = round(shape[0] / goal_size[0])
23     y_compressionrate = round(shape[1] / goal_size[1])
24     compressionrate = x_compressionrate * y_compressionrate
25     for i in range(goal_size[0]):
26         x = x_compressionrate * i
27         for j in range(goal_size[1]):
28             y = y_compressionrate * j
29             # "rechteckiger" Teil des Feldes, welches durch
                   Bilden des Durchschnitts auf ein Element
                   reduziert wird
30             array_slice = array[x:x+x_compressionrate, y:y+
                   y_compressionrate]
31             compressed_array[i,j] = np.uint8(round(np.sum(
                   array_slice) / compressionrate))
32     return compressed_array
33
```

```
34 def customImage(vertical,horizontal):
35     #global custom_img
36     global drawing
37     radius = 38
38     drawing = False
39     global custom_img
40     custom_img = np.zeros((560,560), np.uint8)
41     cv2.namedWindow('Draw a letter')
42     # Echtzeit-Malen
43     while(1):
44         # Mausevent zuweisen + Radius aktualisieren
45         cv2.setMouseCallback('Draw a letter',drawCircle,
               radius)
46         cv2.imshow('Draw a letter',custom_img)
47         # Abfrage der gedrückten Tasten
48         k = cv2.waitKey(1) & 0xFF
49         # Wenn 'Enter'-Taste gedrückt, beende Echtzeit-Malen
50         if k == 13:
51             break
52         # Wenn 'c' gedrückt, setze Bild zurück
53         elif k == ord('c'):
54             custom_img = np.zeros((560,560), np.uint8)
55         # Wenn '+' gedrückt, vergrößere Pinsel
56         elif k == ord('+'):
57             if radius < 100:
58                 radius += 1
59         # Wenn '-' gedrückt, verkleinere Pinsel
60         elif k == ord('-'):
61             if radius > 5:
62                 radius -= 1
63     cv2.destroyAllWindows()
64     # verkleinere Feld auf gewünschtes Format
65     custom_data = compressArray(custom_img,(vertical,
           horizontal))
66     del custom_img
67     return custom_data
```

A.5 Die Klasse „NeuralNetwork“

inspiriert von Raschka, 2017, S.357-360, stark angepasst und erweitert:

```
class NeuralNetwork():
    # 'Konstruktor'
    def __init__(self, structure, activation_functions, epochs, eta,
                 decrease_eta, minibatches, shuffle=True,
                    random_seed=None, autosave=True):
        # Festlegen eines Seeds for den RNG zur Reproduzierbarkeit
        # !! Warnung: Wenn Code verändert wird, kann die Reproduzierbarkeit verloren gehen !!
        np.random.seed(random_seed)
        self.seed = random_seed
        # Anzahl der Neuronen pro Schicht, als Tupel
        self.structure = structure
        # schnellerer Zugriff auf Outputschicht
        self.output_neurons = structure[-1]
        # Definition der Aktivierungsfunktion jeder Schicht, als Tupel
        self.activation_functions = activation_functions
        # Festlegen der Epochenanzahl
        self.epochs = epochs
        # Festlegen der adaptiven Lernrate
        self.eta = eta
        self.decrease_eta = decrease_eta
        # Festlegen, ob Daten regelmäßig gemischt werden sollen
        self.shuffle = shuffle
        # Festlegen der Minibatch-Größe
        self.minibatches = minibatches
        # Festlegen, ob während Lernphase regelmäßig Zwischenspeicherungen vorgenommen werden
        self.autosave = autosave
        # Jeder Eintrag repräsentiert eine Schicht
        # Damit die Indizes identisch zu denen in den Formeln der Arbeit bleiben,
        # gibt es teils dauerhaft leere Einträge für die
```

```
        Inputschicht
29          self.delta = [[]]*(len(structure))
30          self.d_w = [[]]*(len(structure))
31          self.d_b = [[]]*(len(structure))
32          self.net = [[]]*(len(structure))
33          self.o = [[]]*(len(structure))
34          # Initialisierung der Änderungen von Gewicht/Bias
35          for l in range(1,len(self.structure)):
36              self.d_w[l] = np.zeros((self.structure[l],self.
                structure[l-1]))
37              self.d_b[l] = np.zeros(self.structure[l])
38
39      def initializeWeigths(self,w_init=[]):
40          layers = len(self.structure)
41          # Wenn Daten vorhanden, nehme diese, ansonsten
                zufällige Werte
42          if len(w_init) == 0:
43              # Wähle zufällige Werte zwischen -1 und 1 für alle
                    Elemente der Gewichte-Matrizen
44              self.w = [0]
45              for l in range(1,layers):
46                  temp_arr = np.random.uniform(-1.0,1.0,size=
                        self.structure[l]*self.structure[l-1])
47                  temp_arr = temp_arr.reshape(self.structure[l],
                        self.structure[l-1])
48                  self.w.append(temp_arr)
49          elif len(w_init) == layers - 1:
50              self.w = [0]
51              for l in range(layers-1):
52                  temp_arr = np.load(w_init[l])
53                  self.w.append(temp_arr)
54          else:
55              raise ValueError("Please submit no or all weigth-
                    matrices!")
56
57      def initializeBias(self,b_init=[]):
58          layers = len(self.structure)
59          # Wenn Daten vorhanden, nehme diese, ansonsten
```

```
            zufällige Werte
60          if len(b_init) == 0:
61              # Wähle zufällige Werte zwischen -1 und 1 für alle
                    Elemente der Gewichte-Matrizen
62              self.b = [0]
63              for l in range(1,layers):
64                  temp_arr = np.random.uniform(-1.0,1.0,size=
                        self.structure[l])
65                  self.b.append(temp_arr)
66          elif len(b_init) == layers - 1:
67              self.b = [0]
68              for l in range(layers-1):
69                  temp_arr = np.load(b_init[l])
70                  self.b.append(temp_arr)
71          else:
72              raise ValueError("Please submit no or all bias-
                    vectors!")
73
74      # Definiere Aktivierungsfunktionen
75          # logistische Funktion
76      def logisticFunc(self,net,c = 1):
77          return 1 / (1 + np.exp(-c*net))
78          # lineare Funktion
79      def linearFunc(self,net,m = 1):
80          return m * net
81          # Tangens Hyperbolicus
82      def tanHypFunc(self,net,c = 1):
83          return 1 - 2 / (np.exp(2*net*c) + 1)
84
85      # Definiere Ableitung der Aktivierungsfunktionen
86      def logisticGradient(self,net,c = 1):
87          sg = self.logisticFunc(net,c)
88          return sg * (1 - sg)
89      def linearGradient(self,net,m = 1):
90          return net / net
91      def tanHypGradient(self,net,c = 1):
92          sg = self.tanHypFunc(net,c)
93          return 1 - (sg ** 2)
```

```
94
95      def calculateNetmask(self,w,o,b):
96          # Matrix-Vektor-Multiplikation
97          return w.dot(o) + b
98
99      def calculateOutput(self,net,f_act,val=1):
100         # Wähle Aktivierungsfunktion
101         if f_act == "log":
102             return self.logisticFunc(net,val)
103         elif f_act == "tanh":
104             return self.tanHypFunc(net,val)
105         elif f_act == "lin":
106             return self.linearFunc(net,val)
107         else:
108             raise ValueError("{} doesn't represent a proper
                function!".format(f_act))
109
110     def calculateGradient(self,net,f_act,val=1):
111         # Wähle abgeleitete Aktivierungsfunktion
112         if f_act == "log":
113             return self.logisticGradient(net,val)
114         elif f_act == "tanh":
115             return tanHypGradient(net,val)
116         elif f_act == "lin":
117             return linearGradient(net,val)
118         else:
119             raise ValueError("{} doesn't represent a proper
                function!".format(f_act))
120
121     def shuffleBatch(self,dataset,labelset):
122         rng_state = np.random.get_state()
123         np.random.shuffle(dataset)
124         # Damit Label in gleicher Reihenfolge vertauscht
            werden,
125         # muss der RNG-Status auf Stand vor dem Mischen der
            Daten gesetzt werden.
126         np.random.set_state(rng_state)
127         np.random.shuffle(labelset)
```

```
128         return dataset,labelset
129
130     def saveData(self,prefix="xxx",show_location = False):
131         # Speichere Daten mit individuellem, aber für alle
                Daten identischem Präfix
132         struct_string = "-".join(map(str, self.structure))
133         for l in range(1,len(self.structure)):
134             save_string = "{}_layers{}_w{}{}.npy".format(
                    prefix,struct_string,l-1,l)
135             np.save(save_string,self.w[l])
136             save_string = "{}_layers{}_b{}.npy".format(prefix,
                    struct_string,l)
137             np.save(save_string,self.b[l])
138         if show_location:
139             print("Saved as {}_layers{}_XX.npy".format(prefix,
                    struct_string))
140
141     def doFeedforward(self,o_in):
142         # Berechne für gegebenen Netzinput den Netzoutput
143         self.o[0] = self.calculateOutput(o_in,self.
                activation_functions[0])
144         for l in range(1,len(self.structure)):
145             self.net[l] = self.calculateNetmask(self.w[l],self
                    .o[l-1],self.b[l])
146             self.o[l] = self.calculateOutput(self.net[l],self.
                    activation_functions[l])
147
148     def calculateError(self,target):
149         single_errors = target - self.o[-1]
150         total_error = np.sum(np.square(single_errors))/2
151         return single_errors, total_error
152
153     def doBackpropagation(self,single_errors):
154         # Berechne Fehlersignale
155             # Abbruchbedingung
156         self.delta[-1] = self.calculateGradient(self.net[-1],
                self.activation_functions[-1]) * single_errors
157             # Rekursion
```

```
158             for l in range(len(self.structure)-2,0,-1):
159                 self.delta[l] = self.calculateGradient(self.net[l
                        ],self.activation_functions[l]) * np.swapaxes(
                        self.w[l+1],0,1).dot(self.delta[l+1])
160             # Addiere Änderungsfelder für dieses Trainingsmuster
161             for l in range(1,len(self.structure)):
162                 # Berechne dyadische Produkte
163                 self.d_w[l] += np.outer(self.delta[l],self.o[l-1])
164                 self.d_b[l] += self.delta[l]
165
166         def doEpoch(self,current_batch_tr,current_batch_l):
167             t_sum = 0
168             # Gehe jedes Trainingsmuster des Minibatches durch
169             for mb in range(self.minibatches):
170                 # Teste Trainingsmuster im Netz
171                 self.doFeedforward(current_batch_tr[mb])
172                 # Setze Zielwerte (aktuelles Label = 1)
173                 target = np.zeros(self.output_neurons)
174                 target[current_batch_l[mb]] = 1.0
175                 # Berechne Fehler
176                 s, t = self.calculateError(target)
177                 t_sum += t
178                 # Berechne Änderung der Werte des Netzes für
                        dieses Muster
179                 self.doBackpropagation(s)
180             # Berechne durchschnittl. Gesamtfehler
181             t_sum = t_sum / self.minibatches
182             # Update der Gewichtung + Bias
183             for l in range(1,len(self.structure)):
184                 self.w[l] += (self.eta/self.minibatches) * self.
                        d_w[l]
185                 self.b[l] += (self.eta/self.minibatches) * self.
                        d_b[l]
186                 # Zurücksetzen der Änderungsfelder
187                 self.d_w[l] = np.zeros((self.structure[l],self.
                        structure[l-1]))
188                 self.d_b[l] = np.zeros(self.structure[l])
189             return t_sum
```

```
190
191     def Learning(self,training_images,training_labels,w_files
            =[],b_files=[]):
192         amount_training = len(training_labels)
193         if amount_training % self.minibatches != 0:
194             raise ValueError("Size of minibatch must be
                    divisor of available training data!")
195         # Anzahl Epochen um alle Daten 1x durchzugehen
196         epochs_per_dataset = round(amount_training / self.
                minibatches)
197         # Präfix zur Datenspeicherung
198         start_time = time.strftime("%d%m%Y_%H%M%S", time.
                localtime())
199         self.initializeWeigths(w_files)
200         self.initializeBias(b_files)
201         # Liste mit durchschnittlichem Gesamtfehler für jede
                Epoche
202         errors = np.zeros(self.epochs)
203         print("Learning started! Properties of neural net:")
204         struct_string = "-".join(map(str, self.structure))
205         print("Structure: {}\nAutosave: {}\nShuffle: {}\nSeed:
                {}".format(
206             struct_string,self.autosave,self.shuffle,self.seed
                    ))
207         # Starte Timer
208         start = time.clock()
209         last_save = start
210         for n in range(self.epochs):
211             # Erstelle aktuelles Minibatch
212             cur_pos = (n * self.minibatches) % amount_training
                    #TODO !!!!!!!!!!
213             batch_img = training_images[cur_pos:cur_pos + self
                    .minibatches]
214             batch_lab = training_labels[cur_pos:cur_pos + self
                    .minibatches]
215             # Lerne mit Minibatch
216             t_sum = self.doEpoch(batch_img,batch_lab)
217             errors[n] = t_sum
```

```
218             # Update der Lernrate
219             self.eta /= (1 + self.decrease_eta * n)
220             # Überprüfe, ob alle Daten 1x benutzt wurden
221             if n % epochs_per_dataset == 0:
222                 # Anzeige des aktuellen Fortschrittes inkl.
                        verbleibende Zeit
223                 end = time.clock()
224                 passed = (end - start)/60
225                 remaining = (passed / (n+1)) * (self.epochs -
                        n - 1)
226                 print("Calculated epoch {:5d} with average
                        error {:5.4f} | Time passed: {:5.2f} min (
                        about {:5.2f} min left)".format(n+1,t_sum,
                        passed,remaining))
227                 if self.shuffle:
228                     # Mische alle Trainingsdaten
229                     self.shuffleBatch(training_images,
                            training_labels)
230                 if self.autosave and end - last_save > 120.0:
231                     # Zwischenspeichern (min. nach 2 Minuten)
232                     last_save = time.clock()
233                     self.saveData(start_time)
234         print("\n----------------------------------------\n")
235         print("Successfully finished learning phase after
                {:5.2f} minutes!".format((time.clock() - start)/60)
                )
236         a = input("Do you want to save the data now? (y/n)")
237         if a == "y":
238             self.saveData(start_time,True)
239         return errors
240
241     def testNeuralNet(self,tested_images,tested_labels):
242         amount_testing = len(tested_labels)
243         correct_guesses = 0
244         # Initialisiere Feld, um zu speichern, welche Zeichen
                falsch erkannt wurden.
245         wrong_num = np.zeros(self.output_neurons,dtype=int)
246         for i in range(amount_testing):
```

```
247                 guessed_number, certainty = self.
                        categorizeCharacter(tested_images[i])
248                 if guessed_number == tested_labels[i]:
249                     correct_guesses += 1
250                 else:
251                     wrong_num[tested_labels[i]] += 1
252             # Genauigkeit des Netzes entspricht Anteil der richtig
                    klassifizierten Werte
253             accuracy = correct_guesses/amount_testing*100
254             print("The network guessed {} out of {} entries
                    correct!".format(correct_guesses,amount_testing))
255             print("Accuracy: {:4.2f}%".format(accuracy))
256             print("Distribution of errors:",wrong_num)
257             return accuracy
258
259         def categorizeCharacter(self,input_data):
260             self.doFeedforward(input_data)
261             # Gebe Label mit größtem Wert des Outputs zurück
262             network_guess = np.argmax(self.o[-1])
263             # Sicherheit des Netzes = Größter Wert
264             certainty = self.o[-1][network_guess]
265             return network_guess, certainty
```

A.6 Benutzung der Klasse „NeuralNetwork“

eigene Implementierung:

```
1 training_images, training_label, rows, columns = setImageArray
      ("train-images.idx3-ubyte","train-labels.idx1-ubyte")
2 training_label = training_label
3 NumberRecognition = NeuralNetwork((rows*columns,50,10),("lin",
      "log","log",),2000,1.0,0.000003,10000,shuffle=False,
      random_seed = 230801)
4 # Dieses Schema geht von identischen Präfixen für Daten
      derselben Lernphase aus
5 w_files = []
6 b_files = []
7 # String ändern, um Daten zu laden!
8 prefix = ""
```

```
9  if prefix != "":
10     for l in range(1,len(NumberRecognition.structure)):
11         w_files.append(prefix + "w{}{}.npy".format(l-1,l))
12         b_files.append(prefix + "b{}.npy".format(l))
13 # Starte Lernphase
14 errors = NumberRecognition.Learning(training_images,
       training_label ,w_files, b_files)
15
16 # Plotte Gesamtfehler-Graph
17 plt.plot(errors)
18 plt.ylabel("average epoch error")
19 plt.xlabel("epoch")
20 plt.yscale("linear")
21 plt.grid(True)
22 plt.show()
23
24 # Teste Netz mit Überprüfungsset
25 test_images, test_label, rows, columns = setImageArray("t10k-
       images.idx3-ubyte","t10k-labels.idx1-ubyte")
26 test_label = test_label
27 test_acc = NumberRecognition.testNeuralNet(test_images,
       test_label)
28 # Bestimme Überanpassung durch Abweichung der Genauigkeiten
       vom Trainingsset
29 training_acc = NumberRecognition.testNeuralNet(training_images
       , training_label)
30 print("Deviation of accuracy: {:4.2f}%".format(training_acc-
       test_acc))
31
32 # eigenes Zeichnen
33     # Tauschen der Achsen (notwendig für EMNIST)
34 switch_axes = False
35 while(1):
36     # erhalte Feld mit eigenem Zeichen
37     custom_array = customImage(28,28)
38     if switch_axes:
39         custom_array = custom_array.T
40     # Feld in geeignete Form (1D) bringen
```

```
41      custom_array = custom_array.reshape(28*28)
42      # Zeichen mit dem Netz bestimmen
43      guessed_number, certainty = NumberRecognition.
            categorizeCharacter(custom_array)
44      print("Your character is '{}'\nCertainty: {:4.2f}%".format
            (guessed_number,certainty*100))
45      # Wenn leere Zeichenfläche, dann breche ab
46      if np.array_equal(custom_array,np.zeros(28*28)):
47          break
```

Für die Buchstabenerkennung ändert sich der letzte Teil des Codes wie folgt:

```
1 # Wörterbuch, welches jedes Label einem Buchstaben zuordnet
2 label_to_char = {0:'a/A',1:'b/B',2:'c/C',3:'d/D',4:'e/E',5:'f/
      F',6:'g/G',7:'h/H',8:'i/I',9:'j/J',10:'k/K',11:'l/L',12:
3                 'm/M',13:'n/N',14:'o/O',15:'p/P',16:'q/Q',17:'r
                      /R',18:'s/S',19:'t/T',20:'u/U',21:'v/V',22:'
                      w/W',23:'x/X',24:'y/Y',25:'z/Z'}
4 switch_axes = True
5
6 # Anpassung des Outputs
7       print("Your character is '{}'\nCertainty: {:4.2f}%".format
            (label_to_char[guessed_number],certainty*100))
```

B Durchgeführte Tests

Alle Trainingsphasen sind unter Windows 7 und Python 3 abgelaufen. Dabei kam ein Intel(R) Core(TM) i5-7500 CPU @ 3.40 GHz (4-Core) zum Einsatz.

Set	**Netzstruktur (Nr.)**	**Phase**	**Epochen**	**Mini-batch**	**Lern-rate**	**Lern-abstieg**	**Seed**
MNIST	728 lin 50 log 10 (I)	1	600	10000	1.0	2.5	230801
		2	600	10000	0.8	8.0	230801
		3	600	10000	0.8	5.0	230801
		4	600	10000	0.7	3.0	230801
	728 lin 100 log 10 (II)	1	2000	6000	1.0	0.8	230801
	728 lin 80 log 40 log 10 (III)	1	2000	10000	1.0	0.8	230801
EMNIST	728 lin 400 log 100 (IV)	1	2000	6240	1.0	0.8	38952
		2	3800	6240	0.7	0.3	38952
		3	3000	6240	0.6	0.1	38952
		4	1500	6240	20.0	2.5	38952
		5	300	6240	2.0	50	38952

Tabelle 3: Rahmenbedingungen der getesteten Netze. Beachte:
Bei Netzen mit mehreren „Phasen“ wurde von der vorigen Phase aus weitertrainiert.
`shuffle` war dauerhaft ausgestellt
„Lernabstieg“ (mal 10^{-6} !) bezeichnet den Wert, der in die Formel zur Abnahme der Lernrate eingesetzt wird

Netz	**Phase**	**Dauer (min)**	**Genauigkeit (Test)**	**Genauigkeit (Training)**	**Differenz**
(I)	1	53.2	73.30%	73.59%	0.29%
	2	15.4	77.62%	77.97%	0.35%
	3	15.6	80.48%	81.00%	0.52%
	4	15.4	81.73%	82.48%	0.75%
(II)	1	46.6	83.31%	84.56%	1.25%
(III)	1	47.2	83.48%	84.49%	1.01%
(IV)	1	378	25.27%	26.69%	1.43%
	2	371	34.18%	36.42%	2.23%
	3	545	46.38%	49.16%	2.78%
	4	302	70.44%	72.90%	2.46%
	5	66	70.64%	73.70%	3.06%
X	?	??	94.14%	96.15%	2.01%

Tabelle 4: Ergebnisse der getesteten Netze.
X ist ein Netz für Ziffernerkennung aus einer früheren Version des Programms, aufgrund dessen nur noch die Gewichtsmatrizen/Biasvektoren vorhanden sind.

ERKLÄRUNG

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

München, den …06.11.2018…. _____

Unterschrift des/der Schülers/in