Theorie künstlicher neuronaler Netze und Anwendung am Beispiel von Bilderkennung

Facharbeit im Rahmen des Mathematikunterrichts

Lennart Kloock

Albert-Schweitzer-Gymnasium Hürth

Fachlehrerin: Frau Marschner

Eingereicht am 29. März 2021

Inhaltsverzeichnis

1 Einleitung

Der Mensch ist sehr gut darin Objekte, Motive und Zeichen zu erkennen, die auf einem Bild abgebildet sind. Das menschliche Gehirn kann dabei auch Bildern die gleiche Bezeichnung zuordnen, die sich von allen anderen Bildern unterscheiden, die es jemals gesehen hat. Beispielsweise können Menschen eine fremde Handschrift lesen, obwohl sie diese zuvor noch nie gesehen haben. Ein Computer kann dagegen nur festen Programmabläufen folgen und ist allein mit vorbestimmten Algorithmen nicht in der Lage, unbekannte Daten zu analysieren und zu interpretieren. Künstliche neuronale Netze versuchen, dem Computer das Prinzip des Lernens beizubringen. Auf dieses Prinzip wird in dieser Arbeit eingegangen, indem erst der Aufbau und die Funktionsweise eines solchen künstlichen neuronalen Netzes erklärt werden. Anschließend geht es um die dadurch gewonnene Lernfähigkeit und dessen Anwendung. Das Ziel dieser Arbeit ist es, die Theorie und Anwendung von einfachen künstlichen neuronalen Netzen zu erklären und an einem Beispiel der Bilderkennung zu zeigen.

Künstliche Intelligenz ist für die meisten Menschen schon ein Teil ihres Alltags geworden, ohne dass sie es gemerkt haben. Von der Autokorrektur der Handytastatur bis hin zu Sprach-, Bild- und Gesichtserkennung funktioniert heutzutage vieles mit künstlicher Intelligenz und künstlichen neuronalen Netzen. Inzwischen setzen viele Menschen auf die Technik, die durch die neuronalen Netze ermöglicht wird. Diese vielen verschiedenen Anwendungsmöglichkeiten und deren mathematischer Hintergrund sind der Grund für mein Interesse an diesem Thema.

2 Die Geschichte künstlicher neuronaler Netze

Die Forschung an künstlichen neuronalen Netzen hatte bereits in den 1940er Jahren ihre Anfänge und wurde durch viele verschiedene Wissenschaftler betrieben.

1943 veröffentlichten Warren McCulloch und Walter Pitts eine Arbeit[1] in der sie behaupteten, dass die von ihnen beschriebenen Netze so gut wie jede gesuchte Funktion berechnen könnten. Kurz danach nannten sie eine mögliche Anwendungsmöglichkeit im Gebiet der Erkennung von räumlichen Mustern.

In den Jahren nach der Veröffentlichung McCulloch und Pitts' Arbeit wurde viel an den Netzen geforscht und auch verändert. Unter anderem wurde 1949 vom Psychologen Donald O. Hebb die klassische *Hebb'sche Lernregel* aufgestellt. Diese Regel gilt als Grundlage für die meisten anderen Lernregeln und ist nicht nur auf künstliche neuronale Netze anwendbar.

[1] McCulloch und Pitts, „A logical calculus of the ideas immanent in nervous activity."

Frank Rosenblatt und Charles Wightman begannen 1957 am MIT mit der Entwicklung des *Perceptron*s, ein einfaches neuronales Netzwerk, das sie in Form des Neurocomputers *Mark I Perceptron* implementierten. Der *Mark I Perceptron* konnte bereits eine Ziffer in einem 20×20 Pixel Bild erkennen.

1960 entwarfen Bernard Widrow und Marcian E. Hoff die *Deltaregel* beziehungsweise den *LMS-Algorithmus*, mit dem das ebenfalls von Widrow und Hoff entworfene *ADALINE* (*Adaptive Linear Neuron*) arbeitete. Es ist ein System, das bereits kommerziell als Echo-Unterdrückung in Analogtelefonen eingesetzt wurde.

Als 1969 Marvin Minsky und Seymour Papert eine mathematische Analyse des *Perceptron*s veröffentlichten, kam die Forschung im Bereich der künstlichen Intelligenz vollständig zum Erliegen. Sie hatten in ihrem Buch über Frank Rosenblatts einfaches *Perceptron*-Modell einige schwerwiegende Mängel festgestellt, woraufhin viele Forschungsgelder gestrichen wurden. Wegen der knappen Gelder wurde nur noch wenig am Thema weitergeforscht, was Paul Werbos 1974 dazu veranlasste, das schon vorher bestehende *Backpropagation of Error*-Lernverfahren auf neuronale Netze anzuwenden. Dieses Verfahren wurde jedoch erst 10 Jahre nach seiner Veröffentlichung so wichtig, dass es an weiterer Aufmerksamkeit gewann.

Ab den 1980er Jahren nahm der amerikanische Wissenschaftler John Hopfield sehr viel Einfluss auf das mathematisch-informatisch geprägte Thema, unter anderem indem er 1982 seine *Hopfieldnetze* vorstellte. Die *Hopfieldnetze* sind eine Art von neuronalen Netzen, die sich an den Gesetzen des Magnetismus in der Physik orientieren. Außerdem veröffentlicht Hopfield 1985 einen Artikel, in dem er eine Lösung des *Travelling Salesman Problem*s mithilfe seiner *Hopfieldnetze* beschreibt. Minskys Vorhersagen von 1969 stellten sich als größenteils falsch heraus und das Thema gewann die anfängliche Aufmerksamkeit zurück. Von 1980 bis heute hat sich das Thema der künstlichen Intelligenz rasant weiterentwickelt und ist heute bedeutsamer als je zuvor.[2]

[2] Kriesel, *Ein kleiner Überblick über Neuronale Netze*, Abschnitt 1.2.

3 Aufbau und Funktionsweise eines künstlichen neuronalen Netzes

Dieser Abschnitt geht auf den Aufbau und die Arbeitsweise eines künstlichen neuronalen Netzes (im Folgenden KNN) ein.

3.1 Neuronale Netze des Gehirns

Die KNN sind, wie ihr Name schon sagt, inspiriert von den biologischen neuronalen Netzen des Gehirns. Sie sind ein Versuch, diese biologischen Netze auf Basis von Gehirnforschung zu imitieren. Das Gehirn besteht unter anderem aus sogenannten Nervenzellen und deren Verbindungen untereinander. Jede Nervenzelle hat mehrere Tausend dieser Verbindungen, welche auch Synapsen genannt werden. Die Kommunikation über die Synapsen funktioniert in der Regel mithilfe chemischer Neurotransmitter, die als Botenstoffe zwischen zwei Nervenzellen fungieren. Bestimmte Nervenzellen können mehr oder weniger Einfluss auf eine andere Zelle nehmen, je nachdem wie nah die verbindende Synapse am Zellkörper der Zelle ansetzt. Die neuronalen Netze passen sich fortlaufend an ihre Einflüsse (zum Beispiel visuelle oder akustische) an.
Die Architektur der künstlichen Netze ist eine stark vereinfachte Form der hier beschriebenen biologisch-chemischen Netze im Gehirn.[3]

3.2 Künstliche Neuronen

Die Neuronen im Sinne der KNN sind gegenüber den biologischen Neuronen so weit vereinfacht, dass sie nur noch einen Zahlenwert speichern. Sie stellen die kleinste Einheit eines KNN dar und repräsentieren einen Wert, der entweder direkt aus einem Datensatz kommt (zum Beispiel die Graustufe eines Pixels) oder aus einer Berechnung auf Basis der Werte von anderen Neuronen (siehe Unterabschnitt 3.5).[4]

3.3 Ebenen

Die Neuronen der KNN liegen aber nicht, wie im Gehirn, unsortiert vor, sondern sind in Ebenen (engl. *layers*) angeordnet. Ein KNN kann beliebig viele Ebenen von je beliebig vielen Neuronen haben. Das ist unter anderem der Grund für die große Anpassbarkeit der Netzwerke. Eingabeneuronen heißen die Neuronen, die sich in der ersten Ebene eines KNN befinden und deren Werte die Eingabedaten repräsentieren. Die Neuronen, die sich auf der letzten Ebene des Netzes befinden, heißen Ausgabeneuronen und repräsentieren

[3] Trinkwalder, „Netzgespinste“.
[4] 3Blue1Brown, *But what is a Neural Network? — Deep learning, chapter 1*.

die Ausgabe eines Netzes. Außerdem können die künstlichen Netze Ebenen an Neuronen beinhalten, die weder aus Eingabeneuronen noch aus Ausgabeneuron bestehen. Eine solche Ebene wird verdeckte Ebene (engl. *hidden layer*) genannt. Während im Gehirn theoretisch jedes Neuron mit beliebig vielen anderen Neuronen verbunden sein kann, kann ein künstliches Neuron nur mit denen der benachbarten Ebenen verbunden sein.[5]

3.4 Gewichtungen

Die Verbindungen zwischen den künstlichen Neuronen heißen Gewichtungen (engl. *weights*) und stehen repräsentativ für die Synapsen im Gehirn. Jedes Neuron ist durch eine Gewichtung mit allen Neuronen der vorherigen und nachfolgenden Ebene verbunden. Aber auch die Gewichtungen wurden gegenüber den Synapsen stark vereinfacht und auf ihre wichtigste Eigenschaft reduziert. Der Wert des Neurons, von welchem die Gewichtung ausgeht, wird gewichtet an das Neuron, mit dem die Gewichtung verbunden ist, weitergereicht. Der Faktor mit dem der eingehende Wert gewichtet wird, ist veränderbar und eine der wichtigsten anpassbaren Faktoren für den Lernprozess (siehe Abschnitt 4).[6]

3.5 Das Feedforward-Verfahren

Das Weiterverarbeiten einer Menge an Eingabewerten zu einer Menge an Ausgabewerten wird Feedforward (dt. *Vorwärtskopplung* oder *Vorsteuerung*) genannt.
Für ein KNN mit N^l vielen Neuronen auf Ebene l sei z_n^l der Wert des n-ten Neurons auf Ebene l. Der Faktor der Gewichtung, die das n-te Neuron auf Ebene l mit dem p-ten Neuron auf Ebene $l-1$ verbindet, heißt w_{np}^l. Alle Gewichtungen eines KNN seien definiert durch $\overline{w} = (w_{11}^2, w_{12}^2, w_{21}^2, \dots)$. Die Werte der Neuronen der ersten Ebene werden nun bis zur letzten Ebene an Neuronen durchgegeben und dabei folgendermaßen verändert:

- Der Wert eines Neurons entspricht immer der Summe an gewichteten Werten der Neuronen der vorherigen Ebene.
- Ein Wert eines Neurons wird gewichtet, indem er mit dem Faktor der Gewichtung multipliziert wird.

Daraus resultiert folgende von $\overline{w}$ abhängige Funktion, die im weiteren Verlauf dieser Arbeit noch ergänzt und abgewandelt werden wird.

$$z_n^l = \sum_{i=1}^{N^{l-1}} z_i^{l-1} \cdot w_{ni}^l \tag{1}$$

[5] 3Blue1Brown, *But what is a Neural Network? — Deep learning, chapter 1*.
[6] 3Blue1Brown, *But what is a Neural Network? — Deep learning, chapter 1*.

Mit diesem Ausdruck kann nun jeder Wert jedes Neurons berechnet werden. Zuerst wird der Ausdruck auf die Neuronen der zweiten Ebene angewendet, welche die gewichteten Eingabedaten der ersten Ebene benutzen. Nachdem die Werte für die zweite Ebene feststehen, können die der dritten Ebene berechnet werden und so weiter. Nach Anwendung des Feedforward-Verfahrens sind die Werte der letzten Ebene die Ausgabedaten, die das Netz für die Eingabedaten in der ersten Ebene errechnet hat.[7]

3.6 Die Aktivierungsfunktion

Doch allein Ebenen von Neuronen und Gewichtungen reichen nicht, damit ein KNN funktioniert. Um einen ungünstigen Einfluss extremer Werte zu vermeiden, müssen diese normalisiert werden. Diese Normalisierung wird Aktivierung genannt und wird mit einer sogenannten Aktivierungsfunktion umgesetzt. In der Regel versucht eine Aktivierungsfunktion den Wert eines Neurons auf einen Bereich zu begrenzen, zum Beispiel mittels stauchen. Es gibt mehrere Funktionen die dafür infrage kommen, aber die klassische Funktion um diesen Effekt zu erzielen ist die logistische Funktion *sig* (Abbildung 1). Diese Sigmoidfunktion staucht Werte in den Bereich zwischen 0 und 1, indem die Werte für $t \to -\infty$ gegen 0 laufen und für $t \to \infty$ gegen 1 laufen. Dazwischen verläuft der Graph der Funktion S-förmig. Die Funktion wird durch folgenden Term beschrieben.

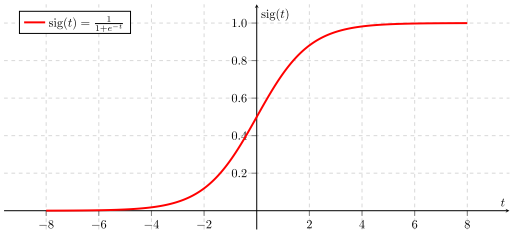


Abbildung 1: Die logistische Kurve

$$sig(t) = \frac{1}{1 + e^{-t}}$$

Für die Aktivierungsfunktion φ gilt im weiteren Verlauf dieser Arbeit immer $\varphi = sig$. Die Aktivierungsfunktion wird auf den Wert eines Neurons (definiert durch (1)) angewendet. Daraus ergibt sich folgende Funktion für den aktivierten Wert a_n^l des n-ten Neurons auf Ebene l.

$$a_n^l = \varphi(z_n^l) \tag{2}$$

Der aktivierte Wert a_n^l wird nun bei Anwendung des Feedforward-Verfahrens anstatt der gewichteten Summe z_n^l genutzt. Deshalb ändert sich (1) nun zu folgendem Ausdruck.[8]

$$z_n^l = \sum_{i=1}^{N^{l-1}} a_i^{l-1} w_{ni}^l \tag{3}$$

[7] 3Blue1Brown, *But what is a Neural Network? — Deep learning, chapter 1*.
[8] 3Blue1Brown, *But what is a Neural Network? — Deep learning, chapter 1*.

3.7 Biases

Auch wenn Neuronen, Gewichtungen und eine Aktivierungsfunktion schon reichen damit ein KNN funktioniert, braucht es in den meisten Fällen noch mehr Variabilität. Aus diesem Grund gibt es den sogenannten Bias (dt. *Verzerrung* oder *Voreingenommenheit*). Auch dieser besteht aus einem veränderbaren Zahlenwert, der für jedes Neuron in einem Netz unterschiedlich sein kann. Ein Bias kann direkten Einfluss auf die Bedeutung seines Neurons nehmen, indem es den Wert direkt erhöht oder erniedrigt. Dies macht das Netz anpassungsfähiger, was beim Trainieren (siehe Abschnitt 4) des Netzes helfen kann. Falls sich das Neuron, zu dem der Bias gehört, nicht in der ersten Ebene des KNN befindet, wird sein Wert auf den bisherigen Wert des Neurons addiert. Neuronen auf der ersten Ebene besitzen keine Biases, da sie die originalen Eingabewerte nur verfälschen würden und dadurch zu ungenaueren Ergebnissen beitragen würden.
b_n^l entspricht dem Bias des n-ten Neurons auf Ebene $l > 1$. Außerdem werden die Funktionen aus (1) und (2) erneut leicht angepasst und sind dadurch nun abhängig von $\overline{b}$, was alle Biases eines Netzes repräsentiert und definiert ist durch $\overline{b} = (b_1^2, b_2^2, \dots)$.[9]

$$z_n^l = b_n^l + \sum_{i=1}^{N^{l-1}} a_i^{l-1} w_{ni}^l \tag{4}$$

$$a_n^l = \varphi(z_n^l) \tag{5}$$

3.8 Beispiel

Abbildung 2 zeigt ein Beispiel für ein KNN mit 6 Neuronen und 9 Gewichtungen, die auf 3 Ebenen verteilt sind. Dabei sind die Eingabeneuronen grün dargestellt, das Ausgabeneuron in Gelb und die Gewichtungen als schwarze Linien. Außerdem beinhaltet das dargestellte Netz eine verdeckte Ebene, dessen Neuronen blau eingefärbt sind. Die Abbildung zeigt die Werte des Netzes, nachdem das Feedforward-Verfahren mit den Werten der ersten Ebene angewendet wurde. Der Einfachheit halber wurde in der Abbildung Gleichung 1 verwendet, was bedeutet, dass das Feedforward-Verfahren ohne Aktivierungsfunktion und Biases angewendet wurde.

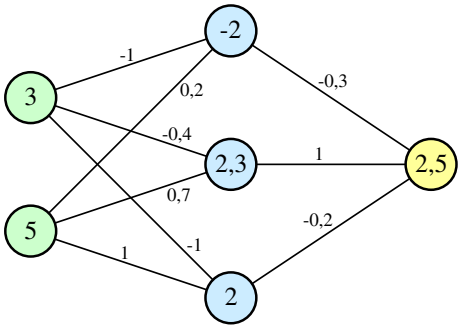


Abbildung 2: Vereinfachte Darstellung eines künstlichen neuronalen Netzes

[9] 3Blue1Brown, *But what is a Neural Network? — Deep learning, chapter 1*.

4 Der Trainingsprozess

In dem in den vorherigen Abschnitten beschriebenen Modell lassen sich bereits Daten eingeben und mithilfe des Feedforward-Verfahrens Ausgabedaten errechnen. Doch passende Ausgabedaten lassen sich nur errechnen, wenn auch alle Gewichtungen und Biases auf das Problem, das das künstliche neuronale Netz lösen soll, angepasst sind. Das gezielte Verändern dieser Variablen wird Trainieren eines KNNs genannt. Während des Trainier- bzw. Lernprozesses wird das KNN schrittweise immer besser darin, Eingabedaten die richtigen Ausgabedaten zuzuordnen.[10]

4.1 Berechnung der Kosten

Um ein KNN zu trainieren, ist es essenziell zu wissen, wie gut die Ergebnisse des Netzes sind. Indem das Feedforward-Verfahren auf Eingabedaten mit bekannten Ausgabedaten angewendet wird, lässt sich errechnen, wie stark sich das Ergebnis des Netzes vom eigentlich erwarteten Ergebnis unterscheidet. Diese Differenz wird Fehler oder auch Kosten genannt und stellt dar, wie sehr sich das Netz mit seinem errechneten Ergebnis irrt. Gegeben seien die Eingabe- und dazu passenden Ausgabedaten d mit

$$d = ((x_1, x_2, \ldots, x_{N^1}), (y_1, y_2, \ldots, y_{N^L})),$$

wobei x_n den Eingabewert für das n-te Neuron in der ersten Ebene darstellt und y_n den n-ten, zu den Eingabedaten passenden, Ausgabewert in der letzten Ebene (L-te Ebene). Die Kosten eines KNN mit L Ebenen lassen sich nun als eine von d, $\overline{w}$ und $\overline{b}$ abhängige Funktion definieren. Die Abhängigkeit der Kosten von d, $\overline{w}$ und $\overline{b}$ drückt sich im folgenden Ausdruck durch a_n^l aus, da der Wert von a_n^l aus diesen Abhängigkeiten resultiert.[11]

$$K_d = \sum_{i=1}^{N^L} (a_i^L - y_i)^2 \tag{6}$$

4.2 Gradientenverfahren

Während des Trainingsprozesses werden Gewichtungen und Biases gezielt verändert, um die durchschnittlichen Kosten möglichst an ein Minimum anzunähern. Da die Kostenfunktion aber viele Funktionsargumente besitzt (alle Gewichtungen und Biases des Netzes), kann nicht so einfach ein Minimum der Kosten gefunden werden. Die klassischen Verfahren zum Bestimmen von Minima sind auf eine so komplexe Funktion wie die Kostenfunktion nicht anwendbar, da sie oft von Millionen von Gewichtungen und Biases abhängig ist. Um

[10] 3Blue1Brown, *Gradient descent, how neural networks learn — Deep learning, chapter 2*.
[11] 3Blue1Brown, *Gradient descent, how neural networks learn — Deep learning, chapter 2*.

ein lokales Minimum dieser Funktion zu finden, wird stattdessen das sogenannte Gradientenverfahren angewendet, welches unabhängig von der Anzahl an Funktionsargumenten funktioniert. Das Gradientenverfahren besteht darin, ein Minimum der Kostenfunktion durch schrittweise Annäherung an dieses zu finden. Um einen den Wert zu finden, um den jedes der Funktionsargumente geändert werden muss, um eine Verringerung der Kosten zu bewirken, wird die partielle Ableitung $\frac{\partial K_d}{\partial w^l_{np}}$ bzw. $\frac{\partial K_d}{\partial b^l_n}$ berechnet. Für eine Minimierung der Kosten wird die Gewichtung bzw. der Bias nun um den negativen Wert der jeweiligen partiellen Ableitung geändert. Das Vorzeichen der Ableitung muss umgekehrt werden, da sich der Kostenwert verringern soll. Der Wert Δw^l_{np} bzw. Δb^l_n beschreibt den Wert, um den die Gewichtung bzw. der Bias geändert wird. Die Werte sind deshalb definiert durch die folgenden Ausdrücke.[12]

$$\Delta w^l_{np} = -\frac{\partial K_d}{\partial w^l_{np}} \tag{7}$$

$$\Delta b^l_n = -\frac{\partial K_d}{\partial b^l_n} \tag{8}$$

4.3 Fehlerrückführung

Um die partielle Ableitung der Kostenfunktion nach einem der Funktionsargumente zu bestimmen, wird die sogenannte Fehlerrückführung (engl. *Backpropagation* oder *Backpropagation of Error*) angewendet. Die Fehlerrückführung startet damit, die partiellen Ableitungen nach den Gewichtungen und Biases (siehe (7) und (8)) der letzten Ebene zu berechnen. Um diese partiellen Ableitungen zu berechnen, wird die Kettenregel genutzt. In der Regel wird eine verkettete Funktion erst wegen der Kettenregel in mehrere Funktionen aufgeteilt, doch im Fall eines KNNs liegen die verschiedenen Kettenglieder bereits als mehrere Funktionen vor. Bei den Kettengliedern handelt es sich um die gewichtete Summe, den aktivierten Wert eines Neurons und die Kostenfunktion. Wenn nun beispielsweise die Ableitung der Kostenfunktion nach einer gewichteten Summe (z^l_n) bestimmt werden muss, kann diese mithilfe der Kettenregel in die partielle Ableitung der Kosten nach der aktivierten gewichteten Summe (a^l_n) und die partielle Ableitung der aktivierten gewichteten Summe (a^l_n) nach der gewichteten Summe (z^l_n) aufgeteilt werden.[13]

[12] 3Blue1Brown, *Gradient descent, how neural networks learn — Deep learning, chapter 2*; Weitz, *Gradientenverfahren (gradient descent) und Fehlerrückführung (backpropagation)*.

[13] 3Blue1Brown, *What is backpropagation really doing? — Deep learning, chapter 3*.

4.3.1 Gewichtungen

Vorerst wird die partielle Ableitung der Kostenfunktion nach einer Gewichtung in zwei Kettenglieder geteilt.

$$\frac{\partial K_d}{\partial w_{np}^l} = \frac{\partial K_d}{\partial z_n^l} \frac{\partial z_n^l}{\partial w_{np}^l} \tag{9}$$

Um $\frac{\partial z_n^l}{\partial w_{np}^l}$ zu berechnen, wird die Funktion z nach w_{np}^l abgeleitet, welches nach Ausdruck (4) in folgendem Ausdruck resultiert.

$$\frac{\partial z_n^l}{\partial w_{np}^l} = a_p^{l-1} \tag{10}$$

Um $\frac{\partial K_d}{\partial z_n^l}$ zu berechnen, wird unterschieden, ob sich das Neuron auf der letzten Ebene oder innerhalb des Netzes befindet. Für den Fall, dass sich das Neuron auf der letzten Ebene befindet, werden die Kettenglieder weiter aufgeteilt und für $\frac{\partial K_d}{\partial z_n^l}$ gilt folgende partielle Ableitung:

$$\frac{\partial K_d}{\partial z_n^L} = \frac{\partial K_d}{\partial a_n^L} \frac{\partial a_n^L}{\partial z_n^L} \tag{11}$$

Die partielle Ableitung der Funktion a nach z_n^l entspricht der Ableitung der Aktivierungsfunktion, was aus (5) hervorgeht.

$$\frac{\partial a_n^l}{\partial z_n^l} = \varphi'(z_n^l) \tag{12}$$

Nach (6) gilt für $\frac{\partial K_d}{\partial a_n^L}$ folgender Ausdruck:

$$\frac{\partial K_d}{\partial a_n^L} = 2(a_n^L - y_n) \tag{13}$$

Falls das Neuron, zu dem die Gewichtung führt, sich nicht auf der letzten Ebene des Netzes befindet, ist die Ableitung nach dieser Gewichtung von den Ableitungen der gewichteten Summen der Neuronen der nachfolgenden Ebene abhängig. Ein Neuron, welches sich nicht in der letzten Ebene befindet, nimmt nicht nur über eine Verbindung Einfluss auf die Kosten, sondern über mehrere Neuronen der nächsten Ebenen. Einfluss bedeutet hier, wie stark Veränderungen im Wert des Arguments sich auf den Wert der Funktion auswirken. Das bedeutet, dass $\frac{\partial K_d}{\partial z_n^l}$ der Summe der einzelnen partiellen Ableitungen der Kostenfunktion nach der gewichteten Summe entspricht.

$$\frac{\partial K_d}{\partial z_n^l} = \sum_{i=1}^{N^{l+1}} \frac{\partial K_d}{\partial a_i^{l+1}} \frac{\partial a_i^{l+1}}{\partial z_i^{l+1}} \frac{\partial z_i^{l+1}}{\partial a_n^l} \frac{\partial a_n^l}{\partial z_n^l} \qquad ,l < L \tag{14}$$

$$= \sum_{i=1}^{N^{l+1}} \frac{\partial K_d}{\partial z_i^{l+1}} \frac{\partial z_i^{l+1}}{\partial a_n^l} \frac{\partial a_n^l}{\partial z_n^l} \tag{15}$$

Hier wird der Vorteil der Fehlerrückführung klar. Da mit der Fehlerrückführung zuerst die Ableitungen der hinteren Ebenen berechnet werden, ist der Wert von $\frac{\partial K_d}{\partial z_i^{l+1}}$ bereits bekannt und kann genutzt werden um weitere Ableitungen der Ebene l zu berechnen. Auch diese

Ableitungen werden benutzt, um die Ableitungen der Ebene davor $(l-1)$ zu berechnen. Ausdruck (11) und (15) lassen sich nun zu folgendem Ausdruck kombinieren.[14]

$$\frac{\partial K_d}{\partial z_n^l} = \begin{cases} \frac{\partial K_d}{\partial a_n^L} \cdot \varphi'(z_n^L) & , l = L \\ \sum_{i=1}^{N^{l+1}} \frac{\partial K_d}{\partial a_i^{l+1}} \frac{\partial a_i^{l+1}}{\partial z_i^{l+1}} \frac{\partial z_i^{l+1}}{\partial a_n^l} \frac{\partial a_n^l}{\partial z_n^l} & , l < L \end{cases} \tag{16}$$

4.3.2 Biases

Um den Wert zu berechnen um den ein Bias b_n^l verändert werden muss, wird ähnlich wie bei den Gewichtungen vorgegangen. Dieser Wert heißt Δb_n^l und ist definiert durch (8). $\frac{\partial K_d}{\partial b_n^l}$ lässt sich mithilfe der Kettenregel in folgende Multiplikation auflösen.[15]

$$\frac{\partial K_d}{\partial b_n^l} = \frac{\partial K_d}{\partial z_n^l} \frac{\partial z_n^l}{\partial b_n^l} \tag{17}$$

Der Wert von $\frac{\partial K_d}{\partial z_n^l}$ ist bereits aus (16) bekannt und die partielle Ableitung von z_n^l nach b_n^l entspricht bei Betrachtung von (4) folgendem Ausdruck.

$$\frac{\partial z_n^l}{\partial b_n^l} = 1 \tag{18}$$

Daraus folgt

$$\Delta b_n^l = -\frac{\partial K_d}{\partial z_n^l} \ . \tag{19}$$

[14] 3Blue1Brown, *Backpropagation calculus — Deep learning, chapter 4*.

[15] 3Blue1Brown, *Backpropagation calculus — Deep learning, chapter 4*.

5 Die Anwendung am Beispiel von Bilderkennung

5.1 Die Problemstellung

Als weitverbreitetes Beispiel zum Training und Testen eines ersten künstlichen neuronalen Netzes wird oft die *MNIST database of handwritten digits* genutzt. Diese Datenbank enthält 70 000 verschiedene Graustufen-Bilder mit 28×28 Pixeln, auf denen jeweils immer eine handgeschriebene Ziffer zwischen 0 und 9 abgebildet ist. Die MNIST Datenbank hat einen Umfang von 60 000 Trainings- und 10 000 Testbildern und deren Bedeutungen als Ziffer. Abbildung 3 zeigt ein Beispiel für ein solches Bild. Diese Bilder soll ein selbst trainiertes künstliches neuronales Netzwerk richtig kategorisieren. Ein KNN wird in der Praxis meist mit anderen Daten trainiert, als mit denen, dessen Genauigkeit getestet wird, da sich so ermitteln lässt, ob das Netz gelernt hat, das gegebene Problem zu lösen oder nur gelernt hat, die Trainingsdaten zu erkennen. Im Rahmen dieser Arbeit wird für diese Aufgabe ein KNN mit $28 \times 28 = 784$ Eingabeneuronen und 10 Ausgabeneuronen trainiert. Das Netz hat 784 Eingabeneuronen, da ein Bild, das erkannt werden soll, 784 verschiedene Pixel hat. Alle Farben eines Pixels liegen zwischen schwarz und weiß, dadurch liegt der Wert eines Eingabeneurons zwischen 0 und 1, wobei alle Farben zwischen schwarz und weiß durch nicht ganze Zahlen zwischen 0 (schwarz) und 1 (weiß) repräsentiert werden. Auf der letzten Ebene des Netzes befinden sich 10 Neuronen, da es 10 verschiedene Ziffern gibt, die ein KNN einem Bild zuordnen kann. Je höher der Wert eines Ausgabeneurons, desto höher ist die Sicherheit, die das Netz in diesen Wert als Ergebnis steckt. Das Ausgabeneuron, welches den höchsten Wert hat, wird als die Entscheidung des Netzes angesehen.[16]

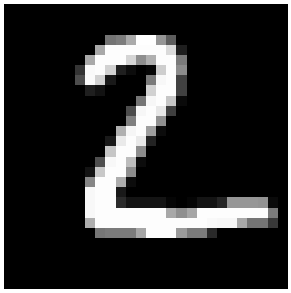


Abbildung 3: Eins der 70 000 Bilder der MNIST Datenbank

5.2 Lernrate

In der Praxis muss das Gradientenverfahren mit einem Faktor reguliert werden, um den Einfluss der mit dem Verfahren ermittelten Werte auf die Variablen des Netzes regulieren zu können. Diese Konstante nennt sich Lernrate (engl. *Learning rate*) und wird als konstanter Faktor η in die Ausdrücke (7) und (19) integriert, sodass die beiden Ausdrücke

[16] 3Blue1Brown, *What is backpropagation really doing? — Deep learning, chapter 3*; LeCun, Cortes und Burges, *The MNIST database of handwritten digits*.

folgendermaßen angepasst werden:

$$\Delta w_{np}^{l} = \eta \cdot -\frac{\partial K_d}{\partial w_{np}^{l}} \tag{20}$$

$$\Delta b_{n}^{l} = \eta \cdot -\frac{\partial K_d}{\partial z_{n}^{l}} \tag{21}$$

Beim Wert der Lernrate gilt es, einen Kompromiss zwischen schnellem, aber ungenauem oder langsamem, aber zuverlässigem Lernen zu finden. Falls die Lernrate zu niedrig eingestellt ist, lernt das Netz gar nicht oder zu langsam. Falls die Lernrate jedoch zu hoch eingestellt ist, wird während der Anwendung des Gradientenverfahren kein lokales Minimum der Kosten gefunden und das Netz lernt ebenfalls kaum oder nur schlecht.[17]

5.3 Berechnen der Genauigkeit

Um die Zuverlässigkeit eines Netzes vergleichen zu können, gibt es den Wert der Genauigkeit. Dieser Wert entspricht dem Anteil an richtigen Zuordnungen, die das Netz getroffen hat. Um die Genauigkeit eines Netzes zu ermitteln, wird das KNN mit den Testdaten getestet und die Anzahl an richtigen Ausgabedaten durch die Anzahl an Testdaten geteilt.

$$\frac{\textit{Anzahl an richtigen Ausgabedaten}}{\textit{Anzahl an eingegebenen Daten}}$$

5.4 Vorgehensweise

Für diesen Teil der Arbeit, habe ich die beschriebenen Verfahren aus Abschnitt 3 und Abschnitt 4 mit der Programmiersprache *Rust* in Form eines eigenen Programms implementiert. Ich habe mich für *Rust* entschieden, da es eine leistungsstarke, schnelle und effiziente Sprache ist. Leistung und Schnelligkeit sind in diesem Kontext ausschlaggebend, da die Dauer des Trainings stark von der Effizienz der Sprache und der meiner Implementierung abhängig ist. Im Folgenden wird ein KNN erstellt, welches Bilder von handgeschriebenen Ziffern erkennen soll. Das von mir dafür verwendete KNN hat zwei Ebenen, was bedeutet, dass es keine verdeckten Ebenen gibt und es nur über eine Eingabe- und Ausgabeebene verfügt. Ein Netz mit nur zwei Ebenen hat sich in der Praxis als eine bessere Wahl als ein Netz mit mehreren Ebenen herausgestellt. Beim Erstellen des KNN wird es mit zufälligen Werten für alle Gewichtungen und Biases initialisiert. Nachdem das Netz initialisiert wurde, wird es mehrmals mit den 60 000 Trainingsbildern und den dazugehörigen Kennzeichnungen trainiert. Ein Training mit allen 60 000 Trainingsbildern wird eine Trainingsepoche genannt. Ein KNN wird in der Regel, wie auch in meiner Implementierung, mit mehreren Hundert bis Tausend Epochen trainiert.

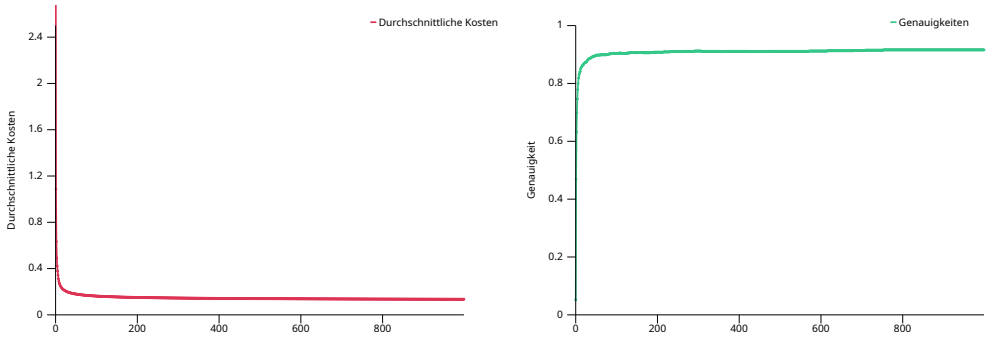
[17] 3Blue1Brown, *Backpropagation calculus — Deep learning, chapter 4*.

5.5 Zuverlässigkeit und Genauigkeit meiner Lösung

Das Trainieren des beschriebenen Netzes in 1 000 Epochen mit einer Trainingsrate von $\eta = 0,001$ hat 47 Minuten und 10 Sekunden gedauert. Diese Zeit könnte jedoch noch verkürzt werden, indem nicht jede Epoche die Genauigkeit gespeichert werden würde (siehe Abbildung 4b). Nachdem das Training beendet ist, hat das Netz eine Genauigkeit von 92,39%, da es 9 239 der 10 000 Testwerte korrekt kategorisiert hat. Das bedeutet in der Praxis, dass das Netz zu 92,39% ein neu eingegebenes Bild, welches es zuvor noch nie gesehen hat, richtig erkennt.

Abbildung 4a zeigt die Entwicklung der Kosten während des Trainingsprozesses. Die x-Achse zeigt die Epochen, während die y-Achse die durchschnittlichen Kosten, die während einer Epoche errechnet wurden, zeigt. Wie man in Abbildung 4a erkennen kann, waren die Kosten anfangs sehr groß und wurden daraufhin schnell niedriger. Dieses Verhalten ist dem Gradientenverfahren zuzuschreiben, da es die Variablen des Netzes bei einer starken Abweichung auch stark anpasst.

Wie bei Abbildung 4a, liegt bei Abbildung 4b die Zahl der Epoche auf der x-Achse. Die y-Achse in Abbildung 4b zeigt die Genauigkeit des Netzes, angegeben als Anteil zwischen 0 und 1. Wie auch bei der Entwicklung der Kosten lässt sich hier zu Anfang ein schnelles Verändern der Werte feststellen.



(a) Entwickelung der Kosten des Netzes während des Trainingsprozesses

(b) Entwickelung der Genauigkeiten des Netzes während des Trainingsprozesses

Abbildung 4

6 Fazit

In Rahmen dieser Facharbeit wurde die Funktionsweise von künstlichen neuronalen Netzen erklärt und in Form eines eigenen Programms implementiert. Die eigene Implementierung kam dabei auf eine Genauigkeit von über 90%, was ich für eine von Grund auf selbst implementierte Lösung für einen sehr guten Wert halte. Das Ziel dieser Arbeit wurde somit erreicht.

Diese Arbeit beschäftigte sich ausschließlich mit einer Art von künstlichen neuronalen Netzen. Um weitere, komplexere Arten von neuronalen Netzen verstehen zu können, ist es essenziell, das Gradientenverfahren, das *Feedforward-Verfahren* und die Fehlerrückführung zu verstehen. Weiterführend könnten die Themen Optimierungsverfahren für den Lernprozess oder die Architektur von KNN interessant sein.

7 Anhang

7.1 Literaturverzeichnis

3Blue1Brown. *Backpropagation calculus — Deep learning, chapter 4*. 2017. URL: `https://youtu.be/tIeHLnjs5U8` (besucht am 21. 03. 2021).

3Blue1Brown. *But what is a Neural Network? — Deep learning, chapter 1*. 2017. URL: `https://youtu.be/aircAruvnKk` (besucht am 23. 02. 2021).

3Blue1Brown. *Gradient descent, how neural networks learn — Deep learning, chapter 2*. 2017. URL: `https://youtu.be/IHZwWFHWa-w` (besucht am 15. 03. 2021).

3Blue1Brown. *What is backpropagation really doing? — Deep learning, chapter 3*. 2017. URL: `https://youtu.be/Ilg3gGewQ5U` (besucht am 28. 03. 2021).

David Kriesel. *Ein kleiner Überblick über Neuronale Netze*. 2007. URL: `https://www.dkriesel.com`.

Yann LeCun, Corinna Cortes und Christopher J.C. Burges. *The MNIST database of handwritten digits*. URL: `http://yann.lecun.com/exdb/mnist`.

Warren S. McCulloch und Walter H. Pitts. „A logical calculus of the ideas immanent in nervous activity.“ Englisch. In: *Bulletin of Mathematical Biology* (1943).

Andrea Trinkwalder. „Netzgespinste“. In: *c't* (2016). URL: `https://www.heise.de/select/ct/2016/6/1458191210995647` (besucht am 16. 03. 2021).

Edmund Weitz. *Gradientenverfahren (gradient descent) und Fehlerrückführung (backpropagation)*. HAW Hamburg. 2017. URL: `https://youtu.be/FE5ROPtWiwE` (besucht am 28. 03. 2021).

7.2 Abbildungsverzeichnis

7.3 Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich diese Facharbeit ohne fremde Hilfe angefertigt habe und nur die im Literaturverzeichnis angegebenen Quellen und Hilfsmittel benutzt habe.

Hürth, 29. März 2021