Technology
Arts Sciences
TH Köln

Technische Hochschule Köln
Fakultät für Informatik und Ingenieurwissenschaften

P R A K T I K U M

Neuronale Netze

Gruppe M.16

Vorgelegt an der TH Köln
Campus Gummersbach
Mathematik 2

ausgearbeitet von:
Maximilian Luca Ramacher
Marius Kühnast Muratcan Garanli
Kai Murza Nick Struckmeyer

Erster Betreuer: Marc Oedingen
Zweiter Betreuer: Peter Wagner

Gummersbach, im Juni 2023

Inhaltsverzeichnis

1 Einleitung Neuronale Netze

Inhalt

Dieses Kapitel liefert einen Einstieg in neuronale Netze. Es werden an Beispielen unterschiedliche Anwendungsfälle anschaulich gemacht. Ebenso soll bereits eine grobe Idee näher gebracht werden, wie ein neuronales Netz aufgebaut ist. Insgesamt werden in diesem Kapitel bereits zahlreiche grundlegende Begriffe erklärt, die in den nachfolgenden Kapiteln benötigt werden.

- Wofür benötigt man Neuronale Netze?
- Einsatzgebiete
- Grobes Prinzip

1.1 Wofür benötigt man Neuronale Netze?

Mit neuronalen Netzen lassen sich Probleme lösen, die mit herkömmlichen Algorithmen nur schwer bis gar nicht lösbar sind. Neuronale Netze sind grundsätzlich in der Lage, nicht lineare Abbildungen anzunähern, Beispiele folgen weiter unten (siehe 1.2). Der große Vorteil neuronaler Netze liegt darin, dass sie selbstständig lernen können, komplexe Muster in großen Datenmengen zu erkennen, wodurch man die entsprechenden Algorithmen nicht mehr von Hand schreiben muss. [CA18]

Aufgrund ihrer Fähigkeit, schnell und effizient komplexe Muster zu erkennen, können neuronale Netze auch Aufgaben lösen, für die man normalerweise Menschen benötigen würde, da herkömmliche Algorithmen diese Probleme nicht zuverlässig genug lösen können oder der Entwicklungsaufwand unverhältnismäßig groß in Relation zum Nutzen wäre. [KSH17]

Häufig sind Muster zu komplex, um sie mit einem klassischen Algorithmus erfassen zu können oder sie sind schlecht von Hand mathematisch formulierbar, wie zum Beispiel die Bilderkennung. Mit Hilfe eines neuronalen Netzes kann man aber dennoch mit relativ geringem Aufwand ein Netz trainieren, das in der Lage ist, solche Bilderkennungsaufgaben zu erledigen [KSH17].

Neuronale Netze sind häufig auch zuverlässiger als klassische Algorithmen, da diese manchmal bestimmte Sonderfälle nicht abdecken, wohingegen neuronale Netze resistenter gegen fehlerhafte Daten und Rauschen sind und besser mit Sonderfällen umgehen können.

1.2 Einsatzgebiete

Neuronale Netze sind äußerst vielfältig und lassen sich für verschiedenste Anwendungen anpassen und trainieren. Zu den häufigsten Anwendungsfällen zählen die Bilderkennung und die Verarbeitung von auditiven Daten.

Die Bilderkennung mittels neuronaler Netze lässt sich beispielsweise dazu einsetzen, medizinische Diagnosen durchzuführen, da diese Netze fähig sind, komplexe Krankheitsbilder zu erkennen [VTY16]. Auch kann die Bilderkennung genutzt werden, um die Position von Personen oder anderen Objekten auf einem Kamerabild zu identifizieren, was zur Realisierung von autonomem Fahren nützlich ist [BMDT16]. Die Bilderkennung mittels neuronaler Netze findet auch in der Industrie Anwendung, um hergestellte Produkte automatisiert auf Mängel zu prüfen und auszusortieren [WCQS18].

Neuronale Netze können auch in Webseiten und Apps eingesetzt werden, um das Nutzerverhalten und Nutzerinhalte zu analysieren und den Nutzern so relevante Inhalte vorschlagen zu können. [YHCE18]

Grundsätzlich sind neuronale Netze in der Lage, beliebige Arten von Eingabedaten zu verarbeiten. So ist es auch möglich, Audio-Daten zu verarbeiten und so beispielsweise Spracherkennung, automatisch generierte Untertitel und Sprachassistenten zu implementieren. [CJLV15]

1.3 Grobes Prinzip

Neuronale Netze sind von der Funktionsweise menschlicher Gehirne inspiriert und versuchen dadurch eine ähnliche Lernfähigkeit zu erzielen. Dabei werden Netze aus Neuronen simuliert, um menschliche Lernprozesse und kognitive Fähigkeiten nachzuahmen. [CA18]

Die Neuronen neuronaler Netze sind in Schichten organisiert, die untereinander verbunden sind, wobei die Signale von Schicht zu Schicht durch das Netz geleitet werden. Jedes einzelne Neuron eines Netzes verarbeitet seine Eingabedaten durch die Anwendung mathematischer Funktionen und leitet das Ergebnis dann an das nächste Neuron weiter. Die Verbindungen zwischen den Neuronen sind von variabler Stärke und sind der primäre Faktor, der beim Lernprozess verändert wird. [CA18]

Neuronale Netze werden trainiert, indem man sie auf einen Trainingsdatensatz anwendet und die Verknüpfungen der Neuronen basierend auf der Stärke der Abweichung vom gewünschten Ergebnis anpasst. Der Trainingsdatensatz enthält dabei beispielhafte Eingabedaten und die dazugehörige gewünschte Ausgabe. [CA18]

Nach dem Trainingsvorgang kann man das Netz dann auf ihm unbekannte Daten anwenden, wobei es dann eine Ausgabe basierend auf den erlernten Mustern bildet. [CA18]

2 Neuronen

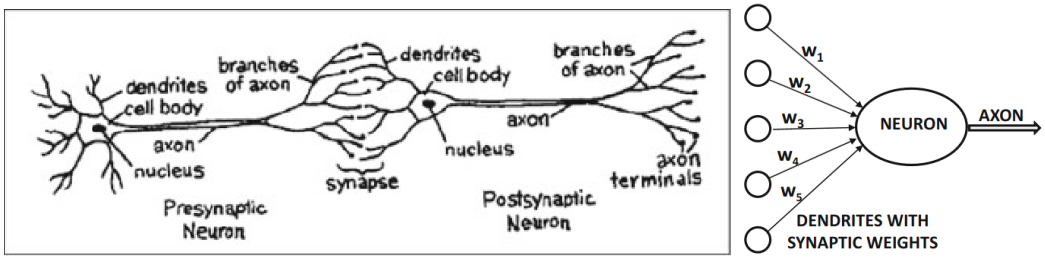
Inhalt

Im letzten Kapitel wurde bereits ein Einblick in den Aufbau eines neuronalen Netzes gegeben. Dieses Kapitel widmet sich Neuronen, deren Bedeutung in einem Netz und wie diese untereinander verknüpft sind. Es wird hier bereits die einfachste Form eines Neuralen Netzwerk vorgestellt.

- Was sind Neuronen
- Arten von Neuronen
- Funktionsweise
- Aktivierungsfunktion
- Schichtenmodell

2.1 Was sind Neuronen?

Das menschliche Nervensystem besteht aus Neuronen, welche mit Axonen oder Dendriten verknüpft sind. Diese Verbindungen werden auch Synapsen genannt. Die variable Stärke der Synapsen ermöglichen das Lernen. Dieser biologische Mechanismus wird durch neurale Netze simuliert.



(a) Synapse (b) Neurales Netzwerk

Abbildung 1: Bild aus dem Buch 'Neural Networks and Deep Learning' von Charu C. Aggarwal

Ein neuronales Netz besteht aus mindestens einem Neuron. Neuronen sind essentielle Bestandteile von neuralen Netzen. Sie nehmen Eingabedaten entgegen und wandeln diese in Ausgabedaten um. Neben den Eingabedaten werden auch Weight-Parameter übergeben,

welche die zu berechnenden Werte beeinflussen. Das eigentliche „Lernen“ erfolgt durch diesen Einfluss.[CA18]

2.2 Arten von Neuronen

2.2.1 Input Neuronen

Input Neuronen erhalten Rohdaten und übergeben diese zusammen mit Weights an die Output Neuronen oder die versteckten Neuronen. Jedes Input Neuron ist mit jedem Neuron aus der nächsten Schicht (Hidden oder Output) verknüpft. Eine Aktivierungsfunktion entscheidet, ob das Neuron in der nächsten Schicht aktiviert werden soll. [CA18]

Bias Neuronen

Das Bias Neuron hat typischerweise den Wert 1. Das Produkt des Bias Neurons und dessen Weight wird auf die Summe der Neuronen aus der Input/Hidden Layer mit aufaddiert. Das heißt, dass dieses Neuron das Ergebnis der Aktivierungsfunktion direkt beeinflussen kann. [CA18]

2.2.2 Versteckte Neuronen

Versteckte Neuronen befinden sich in den versteckten Schichten zwischen der Eingabe- und der Ausgabeschicht. Diese erhalten Werte von Neuronen aus der vorherigen Schicht und übermitteln diese an die Nächste. [CA18]

2.2.3 Output Neuronen

Output Neuronen befinden sich in der letzten Schicht eines neuralen Netzwerks. Auch Output Nodes werden mit Aktivierungsfunktionen aktiviert, wie z.B. der softmax Funktion. [CA18]

2.3 Funktionsweise

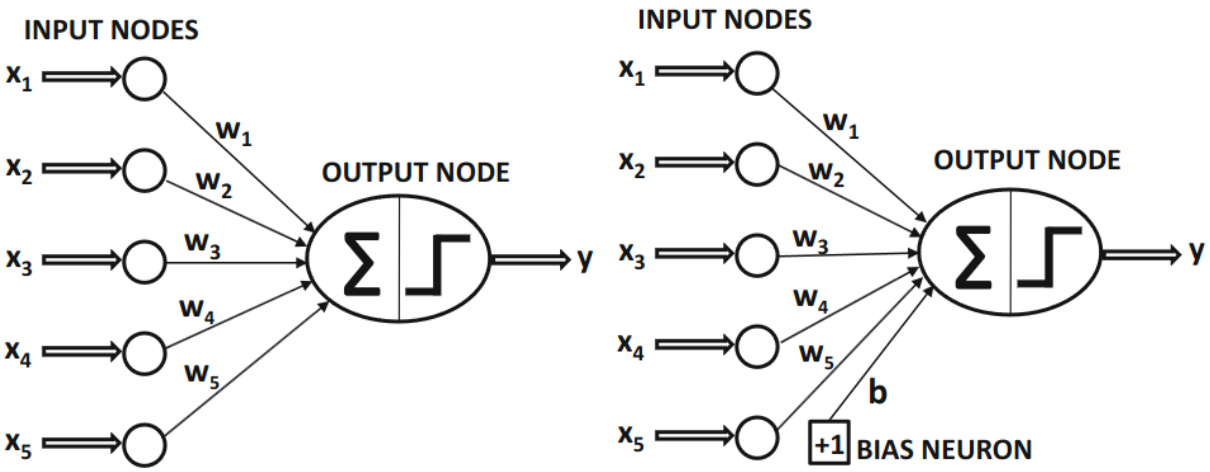
2.3.1 Perceptrons

Die simpelste Form eines Neuralen Netzwerks ist ein Perceptron. Es kann nur binäre Entscheidungen {-1,+1} treffen. Ein Perceptron besteht aus einer Input Layer und einem Output Node. Die Input Layer beinhaltet d nodes, welches d Merkmale $\overline{X} = [x_1...x_d]$ mit Weights $\overline{W} = [w_1...w_d]$ übermittelt. Die lineare Aktivierungsfunktion sign berechnet dann die Vorhersage $\hat{y}$. [CA18]

$$\hat{y} = sign\{\overline{W} \cdot \overline{X}\} = sign\{\sum_{j=1}^{d} w_j x_j\}$$

In vielen Fällen wird ein nicht variables Element b mit in der Rechnung berücksichtigt. Dies verursacht, dass der Mittelwert der Vorhersage nicht 0 ist. [CA18]

$$\hat{y} = sign\{\overline{W} \cdot \overline{X} + b\} = sign\{\sum_{j=1}^{d} w_j x_j + b\}$$



(a) Perceptron ohne Bias (b) Perceptron mit Bias

Abbildung 2: Aufbau von Perceptronen, Bild aus dem Buch 'Neural Networks and Deep Learning' von Charu C. Aggarwal

Durch die sogenannte Minimierung lässt sich der Fehler der Vorhersage verringern. Hierzu werden neben den Features x, auch Labels y in einem Feature-Label Paar eingeführt.

$$\text{Minimize}_{\overline{W}} L = \sum_{(\overline{X},y)\in\mathcal{D}} (y - \hat{y})^2 = \sum_{(\overline{X},y)\in\mathcal{D}} (y - sign\{\overline{W} \cdot \overline{X}\})^2$$

Diese Art der Minimierung wird auch als Loss-Funktion bezeichnet. Die obige Funktion führt zu einer treppenstufigen Loss-Ebene, welche für gradient-descent ungeeignet ist. Um diesem Problem entgegenzuwirken, wird eine Smooth-Funktion angewendet.

$$\Delta L_{\text{smooth}} = \sum_{(\overline{X},y)\in\mathcal{D}} (y - \hat{y})\overline{X}$$

Beim Trainieren eines neuralen Netzwerks werden Eingabedaten $\overline{X}$ einzeln oder in kleinen Batches eingespeist, um die Vorhersage $\hat{y}$ zu generieren. Die Weights werden dann durch den Fehlerwert $E(\overline{X}) = (y - \hat{y})$ aktualisiert.

$$\overline{W} \Rightarrow \overline{W} + \alpha(y - \hat{y})\overline{X}$$

Der Parameter α reguliert die Lernrate des Neuralen Netzwerks. Der Perceptron-Algorithmus durchläuft die Trainingsdaten mehrmals, bis die Weight-Werte konvergieren. Ein solcher Durchlauf wird als Epoche bezeichnet.
Der Perceptron-Algorithmus kann auch als stochastische Gradientenabstiegsmethode betrachtet werden. [CA18]

2.3.2 Aktivierungsfunktion

Aktivierungsfunktionen ermöglichen den Neuronen nicht-lineare Outputs zu produzieren. Außerdem wird durch diese Funktionen entschieden, welche Neuronen aktiviert werden und wie die Inputs gewichtet werden. Zur Notation von Aktivierungsfunktion nutzen wir Φ.

$$\hat{y} = \Phi(\overline{W} \cdot \overline{X})$$

Lineare Aktivierung

Die simpelste Aktivierungsfunktion $\Phi(\cdot)$ ist die lineare Aktivierung. Sie bietet keine nicht linearität. Sie wird oft in Output Nodes verwendet, wenn das Ziel ein reeller Wert ist [CA18].

$$\Phi(v) = v$$

Nicht lineare Aktivierung

Um auch nicht lineare Probleme lösen zu können, werden nicht lineare Aktivierungsfunktionen verwendet. In den frühen Tagen der Entwicklung von neuralen Netzen wurden sign, sigmoid und hyperbolic tangent Funktionen genutzt. Die sign Funktion $\Phi(v) = \text{sign}(v)$ generiert nur binäre {-1,+1} Ausgaben. Aufgrund der Nichtstetigkeit der Funktion, können beim Trainieren keine Loss-Funktionen verwendet werden. Die Sigmoid Funktion $\Phi(v) = \frac{1}{1+e^{-v}}$ generiert Werte zwischen 0 und 1. Sie eignet sich deshalb für Rechnungen die als Wahrscheinlichkeiten interpretiert werden sollen. Der Graph der Tanh Funktion $\Phi(v) = \frac{e^{2v}-1}{e^{2v}+1}$ hat eine ähnliche Form wie die der Sigmoid Funktion. Sie unterscheidet sich jedoch in der Skalierung, denn ihr Wertebereich liegt zwischen -1 und 1. Die Tanh Funktion lässt sich auch durch die Sigmoid Funktion darstellen [CA18].

$$\tanh(v) = 2 \cdot \text{sigmoid}(2v) - 1$$

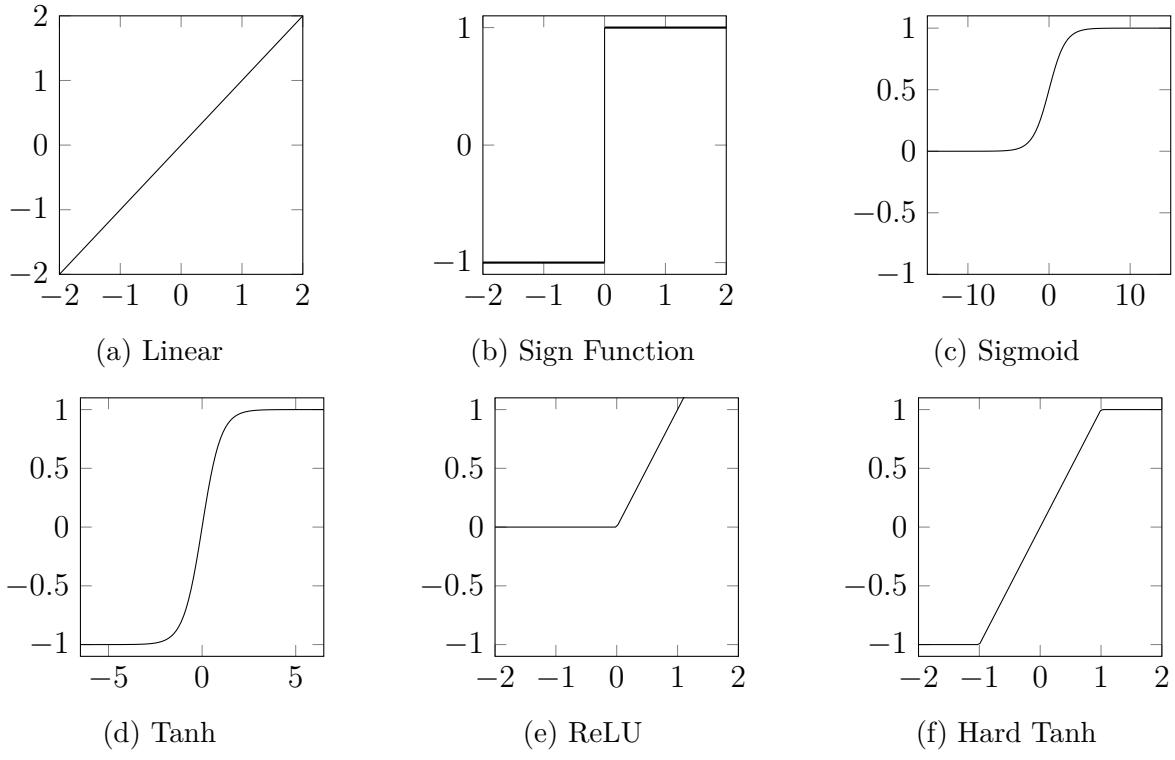
Wenn die Ausgabe der Berechnung positiv sowie negativ sein kann, ist die tanh Funktion der sigmoid Funktion vorzuziehen. Außerdem ist es einfacher zu trainieren, weil die Funktion Mittelwertzentriert und der Gradient größer ist [CA18].

Piecewise lineare Aktivierung

Historisch wurden die Sigmoid und Tanh Funktion zur Einführung von Nichtlinearität genutzt. Heutzutage sind piecewise linear activation Funktionen (Stückweise linear) beliebter, weil diese das Trainieren von mehrschichtigen neuronalen Netzen einfacher machen. Zu den am häufigsten genutzten Aktivierungsfunktionen gehört ReLU (Rectified Linear Unit) $\Phi(v) = \max\{v, 0\}$. Das Ergebnis der ReLU Funktion ist 0, wenn die gewichtete Summe der Inputs v kleiner 0 ist, sonst ist das Ergebnis das unveränderte v. Eine weitere oft genutzte Aktivierungsfunktion ist die hard tanh Funktion $\Phi(v) = \max\{\min[v, 1], -1\}$. Das Ergebnis der hard tanh Funktion ist -1 für $v < 1$ oder 1 für $v > 1$. Sonst ist das Ergebnis das unveränderte v [CA18].

Differenzierbarkeit von Aktivierungsfunktionen

Die meisten neuralen Netze verwenden das Gradienten-Abstiegsverfahren zum Lernen. [CA18] Aus diesem Grund ist die Differenzierbarkeit von Aktivierungsfunkionen besonders wichtig. Die Ableitung der linearen Aktivierungsfunktion ist immer 1. Die Ableitung von $sign(v)$ ist 0 für alle Werte außer $v = 0$. Die fehlende Stetigkeit dieser Funktion ist einer der Gründe, weshalb diese selten in Verlustfunktionen verwendet wird, auch wenn sie als Aktivierungsfunktion verwendet wurde. Alle anderen vorher aufgezählten Aktivierungsfunktionen sind differenzierbar. [CA18]



(a) Linear (b) Sign Function (c) Sigmoid

(d) Tanh (e) ReLU (f) Hard Tanh

Abbildung 3: Aktivierungsfunktionen

2.3.3 Verlustfunktionen

Verlustfunktionen (Loss Functions) oder auch bekannt als Kostenfunktionen bewerten den berechneten Wert $\hat{y}$ eines neuralen Netzes in Relation zum erwarteten Wert y. Um die Genauigkeit der vorhergesagten Werte zu optimieren, werden Verlustfunktionen beim Trainieren verwendet [GBCL18]. Auch wie bei den Aktivierungsfunktionen gibt es viele Verlustfunktionen, welche erheblichen Einfluss auf das Ergebnis eines neuralen Netzes haben. Die Wahl der Verlustfunktionen hängt von der Architektur des neuralen Netzes und der Ausgabewerte der Output-Nodes ab. Für ein einfachen Perceptron wird eine Verlustfunktion der Form $(y - \hat{y})^2$ verwendet. Alternative Verlustfunktionen währen z.B. die hingle-loss-Funktion $L = \max\{0, 1 - y \cdot \hat{y}\}$, welche genutzt werden kann, um z.B. die Support Vector Machine Lernmethode zu implementieren. Für diese Verlustfunktion muss $y \in \{-1, +1\}$ gelten und $\hat{y}$ muss reelle Zahlen darstellen [CA18].
Für probabilistische Vorhersagen, bei denen $y \in \{-1, +1\}$ gilt und $\hat{y}$ eine reelle Zahl ist, eignet sich $L = \log(1 + \exp(-y \cdot \hat{y}))$ als Verlustfunktion. Diese Art von Verlustfunktion implementiert die logisitical Regression Machine Learning Methode [CA18].

2.3.4 Schichtenmodell

Neuronale Netze sind in einer Struktur aus mehreren aufeinanderfolgenden Schichten aufgebaut. Man unterscheidet grundlegend zwischen drei verschiedenen Arten von Schichten, der Eingabeschicht (”input layer”), den versteckten Schichten (”hidden layer”) und der Ausgabeschicht (”output layer”). Die einzige Aufgabe der Eingabeschicht ist es, die Daten der Eingabevariablen darzustellen und an die folgenden Neuronen weiterzugeben, was auch bedeutet, dass diese Neuronen keine Aktivierungsfunktionen verwenden. Die Berechnungen, die in den Hidden-Layers stattfinden sind nicht nach außen hin sichtbar, lediglich die Ausgabe des Output-Layers gelangt nach außen. [CA18]

Die einfachste Variante eines neuronalen Netzes ist das Single-Layer-Perceptron. Dieses besteht lediglich aus einem Input-Layer, der die Eingabedaten in das Netz einspeist und einem Output-Layer, der die Ergebnisse aus dem Netz ausgibt. Es ist also nur eine Schicht an Verbindungen zwischen der Ein- und Ausgabeschicht vorhanden. Diese Art von Netzen ist aber verhältnismäßig wenig leistungsfähig und kann keine komplexen Zusammenhänge verarbeiten und nur Werte zwischen 0 und 1 ausgeben. Es existieren jedoch auch neuronale Netze mit mehreren Schichten, wie zum Beispiel die Multilayer-Perceptrons. Dabei handelt es sich um Netze, die über mindestens einen Hidden-Layer verfügen. Multilayer-Perceptrons und feed-forward neuronale Netze im Allgemeinen sind außerdem wie ein azyklischer Graph aufgebaut. Solche Netze sind fähig, auch komplexere Muster zu erkennen. Durch eine Erhöhung der Anzahl an Schichten kann die Leistungsfähigkeit des Netzes weiter erhöht werden, jedoch steigt auch die benötigte Rechenleistung an. Wenn

ein neuronales Netz über zahlreiche Schichten verfügt, dann wird auch von einem Deep-Neural-Network gesprochen. [CA18]

Eine Variante der neuronalen Netze sind sogenannte Convolutional Neural Networks (CNNs). Diese Netze sind aufgrund der speziellen verwendeten Schichten besonders gut dazu geeignet, Muster in Bildern zu erkennen und eignen sich daher für die Bilderkennung. CNNs setzen vor allem zwei spezielle Arten von Schichten ein, die convolutional layers und die pooling layers. Convolutional layer setzen mehrere Filter Matrizen ein, wodurch besonders gut Muster in 2D Matrizen erkannt werden können, da diese Filter auch immer umliegende Pixel in ihre Berechnungen einbeziehen, was ihnen erlaubt, beispielsweise Kanten und Formen zu erkennen. Die Pooling Layers hingegen dienen dazu, die Eingabe zu downscalen, indem sie benachbarte Merkmale aggregieren. Eine Implementation davon ist MaxPooling2D, ein Pooling Layer, der zweidimensionale Eingabedaten herunterskaliert, indem er aus einem Segment der Eingabe immer nur den maximalen Wert auswählt. Das kann dazu beitragen, dass das Netz nicht overfitted wird und robuster gegen Abweichungen in den Eingabedaten wird. [CA18] [KSH17]

2.4 Wie sind Neuronen miteinander verknüpft

In einem einfachen neuronalen Netz wie einem Multilayer-Perceptron sind alle Neuronen der einen Schicht jeweils mit allen Neuronen der folgenden Schicht verbunden. Jeder Verbindung zwischen zwei Neuronen ist jeweils ein sogenanntes Weight zugeordnet. [CA18]

2.4.1 Weights

Die Verknüpfungen zwischen Neuronen leiten Signale nicht einfach unverändert an das nächste Neuron weiter. Die Weights, die jeder Verbindung zwischen Neuronen zugewiesen sind, können die Signale sowohl verstärken als auch abschwächen, wobei ein höheres Weight zu einer Verstärkung führt. Weights sind der Faktor innerhalb des neuronalen Netzes, der während des Trainingsvorgangs verändert wird. Jedem Neuron ist dabei ein eigenes Weight zugeordnet. Die Gewichtungen dienen also dazu, dass das neuronale Netz überhaupt lernen kann. Unwichtige Verbindungen werden während des Lernprozesses abgeschwächt und einige können dadurch sogar fast gänzlich blockiert werden, wohingegen andere Verbindungen verstärkt werden. Dadurch ist es zum Beispiel möglich, dass ein Neuron nur die Werte von einzelnen anderen Neuronen verarbeitet und andere ignoriert. Dies hilft den Neuronen dabei, Muster in den Eingabedaten zu erkennen. [TR17] [CA18]

3 Gradientenverfahren

Inhalte des *Gradientenverfahren*

Um die Verlustfunktion zu minimieren gibt es verschiedene Optimierungsverfahren. Das wohl bekannteste und am häufigsten eingesetzte Verfahren wird als Gradientenverfahren bezeichnet. Das Kapitel Gradientenverfahren stellt die Grundlagen dar, die für das Verständnis des Lernprozesses eines neuronalen Netzwerks im nachfolgenden Kapitel erforderlich sind.

- Wofür braucht man das Gradientenverfahren?
- Grundkonzepte des Gradientenverfahrens
- Gefährliche Fehlerquellen

3.1 Wofür braucht man das Gradientenverfahren?

Das Gradientenverfahren (engl. gradient descent) wird genutzt, um ein Minimum einer Funktion mit beliebig vielen Parametern / Dimensionen zu finden. Natürlich könnte man nun denken, dass man dies durch bereits bekannte Methoden auch algebraisch berechnen könnte, jedoch wird dies bei einer Funktion mit tausenden oder mehr Parametern sehr schwierig oder gar unmöglich. Für neuronale Netze nutzt man das Gradientenverfahren konkret, um ein Minimum der Verlustfunktion zu bestimmen. Durch diese Berechnung lässt sich mithilfe der Backpropagation jedes einzelne Gewicht und jeder Bias-Wert anpassen, hierdurch "lernt" das neuronale Netz. [TR17]

3.1.1 Was ist der Gradient einer Funktion?

Der Gradient einer Funktion $f(x_1, x_2, ..., x_n)$ ist definiert durch die Funktion $\nabla f(x_0)$, welche den Spaltenvektor V liefert, in welchem jede Komponente v_1 bis v_n die partielle Ableitung der Funktion f nach dem jeweiligen Parameter x_i an der Stelle x_0 darstellt. Konkret also:

$$\nabla f(x_0) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x_0) \\ \frac{\partial f}{\partial x_2}(x_0) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x_0) \end{bmatrix}$$

Der Gradient zeigt die Richtung des steilsten Anstiegs an einem bestimmten Punkt der Funktion. Wenn man in die Richtung des Gradienten geht, erhöht sich die Funktion so schnell wie möglich. Geht man hingegen in die entgegengesetzte Richtung, also Richtung des negativen Gradienten, verringert sich die Funktion am schnellsten. Dieser Aspekt ist entscheidend für das Gradientenverfahren.

Im Folgenden wird der Gradient einer Funktion an einem simplen Beispiel berechnet:

Sei $f(x, y) = x^2 + y^2$, nun ist der Gradient für den Punkt $x = 5, y = 3$ gesucht. Es gilt

$$\frac{\partial f}{\partial x}(x, y) = f_x(x, y) = 2x$$

$$\frac{\partial f}{\partial y}(x, y) = f_y(x, y) = 2y$$

Somit ergibt sich für unsere Funktion f folgender Gradient für den Punkt $x = 5, y = 3$

$$\nabla f(5, 3) = \begin{bmatrix} 2 * 5 \\ 2 * 3 \end{bmatrix} = \begin{bmatrix} 10 \\ 6 \end{bmatrix}$$

[CA18]

3.2 Grundkonzepte des Gradientenverfahrens

3.2.1 Wie funktioniert das Gradientenverfahren?

Das Gradientenverfahren ist ein iteratives Verfahren, bei welchem man sich in jedem Iterationsschritt immer näher in die Richtung des steilsten Abstiegs einer Funktion $f(x_1, x_2, ..., x_n)$ bewegt. Somit nähert man sich nach einigen Iterationen zuverlässig einem Minimum der Funktion f an.

Wie bereits oben erwähnt, gibt uns der Gradientenvektor $\nabla f(x_0)$ einer Funktion $f(x_1, x_2, ..., x_n)$ die Richtung des steilsten Anstieges vom Punkt x_0 aus gesehen. Passen wir also jeden Parameter x_i um den durch den Gradientenvektor gegebenen Wert v_i an, bewegen wir uns damit weiter in Richtung des steilsten Anstieges. Da wir uns beim Gradientenverfahren aber für das Minimum einer Funktion interessieren, geht man stattdessen in die entgegengesetzte Richtung des Gradientenvektors $-\nabla f(x_0)$, also die Richtung des steilsten Abstiegs. Der Gradientenvektor gibt jedoch nicht an, wie weit man in die Richtung des steilsten Abstiegs gehen sollte. Um also zu verhindern, dass man das Minimum 'überschreitet' moderiert man die Schrittweite durch eine sogenannte Lernrate η. Der Startpunkt x_0

muss außerdem zu Beginn des Gradientenverfahrens zufällig ausgewählt werden. Damit haben wir die Grundidee des Gradientenverfahrens:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}_{Neu} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}_{Alt} - \eta \nabla f(x_0)$$

Die Schritte des Gradientenverfahres sind also folgende:

1. Auswahl eines zufälligen Startpunktes / Startparameter x_0
2. Berechnen des Gradientenvektors $\nabla f(x_0)$
3. Anpassen der Startparameter durch den negativen Gradientenvektor multipliziert mit der Lernrate

Die Schritte 2. und 3. wiederholt man nun eine feste Anzahl an Iterationsschritten oder bis die Ursprungsfunktion f gegen einen Wert konvergiert.

Im Folgenden wird das Gradientenverfahren an einem konkreten Beispiel erläutert. Sei $f(x, y) = 3x^2 + 6y^2$ und ein zufällig ausgewählter Startpunkt $x = 3$ und $y = 4$. Die Lernrate setzen wir auf $\eta = 0.05$. Dann berechnet sich der Gradient $\nabla f(3, 4)$ wiefolgt:

$$\frac{\partial f}{\partial x}(x, y) = f_x(x, y) = 6x$$

$$\frac{\partial f}{\partial y}(x, y) = f_y(x, y) = 12y$$

$$\Rightarrow \nabla f(3, 4) = \begin{bmatrix} 6 * 3 \\ 12 * 4 \end{bmatrix} = \begin{bmatrix} 18 \\ 48 \end{bmatrix}$$

Nun passen wir die Startparameter gemäß der Vorschrift mithilfe des negativen Gradientenvektors multipliziert mit der Lernrate an:

$$\begin{bmatrix} 3 \\ 4 \end{bmatrix} - 0.05 * \begin{bmatrix} 18 \\ 48 \end{bmatrix} = \begin{bmatrix} 2.1 \\ 1.6 \end{bmatrix}$$

Bereits jetzt ergibt sich ein erheblicher Unterschied, wohingegen $f(3, 4) = 51$ ergibt, bekommen wir mit unseren aktualisierten Parametern bereits $f(2.1, 1.6) = 28.59$. Wir nähern uns also einem Minimum an! Die weiteren Iterationsschritte sind nur noch in

verkürzter Form angegeben:

$$\begin{bmatrix}2.1\\1.6\end{bmatrix} - 0.05 * \begin{bmatrix}6 * 2.1\\12 * 1.6\end{bmatrix} = \begin{bmatrix}1.47\\0.64\end{bmatrix} \Rightarrow f(1.47, 0.64) = 8.94$$

$$\begin{bmatrix}1.47\\0.64\end{bmatrix} - 0.05 * \begin{bmatrix}6 * 1.47\\12 * 0.64\end{bmatrix} = \begin{bmatrix}1.02\\0.256\end{bmatrix} \Rightarrow f(1.02, 0.256) = 3.51$$

$$\begin{bmatrix}1.02\\0.256\end{bmatrix} - 0.05 * \begin{bmatrix}6 * 1.02\\12 * 0.256\end{bmatrix} = \begin{bmatrix}0.714\\0.1024\end{bmatrix} \Rightarrow f(0.714, 0.1024) = 1.59$$

$$\begin{bmatrix}0.714\\0.1024\end{bmatrix} - 0.05 * \begin{bmatrix}6 * 0.714\\12 * 0.1024\end{bmatrix} = \begin{bmatrix}0.5\\0.04\end{bmatrix} \Rightarrow f(0.5, 0.04) = 0.76$$

Wie man sieht, nähern sich unsere Funktionswerte mit jedem Iterationsschritt der 0. Würde man das Gradientenverfahren einige Iterationen weiter ausführen, so würde man schlussendlich die Werte $x = 0$ und $y = 0$ herausbekommen. Dort liegt unser Minimum.

Die Auswahl des Startpunktes x_0 sowie die Wahl der Lernrate η spielen eine große Rolle beim Erfolg des Gradientenverfahrens, hierauf wird hier jedoch nicht weiter eingegangen. [CA18]

3.3 Gefährliche Fehlerquellen

3.3.1 Steckt man in einem lokalen Minimum fest?

Auf der Suche nach dem globalen Minimum kann der Algorithmus in einem lokalen Minimum enden und somit das Erreichen des globalen Minimums verhindert werden. Ein lokales Minimum tritt auf, wenn das Netzwerk an einem Punkt des Fehlergradienten auf eine niedrigere Fehlerfunktionsebene trifft, aber in der Nähe dieses Punktes ein anderer Punkt mit noch niedrigerem Fehler existiert (siehe Abb. 4). Da neurale Netze häufig große Anzahlen von Parametern haben, kann die Suche nach dem globalen Minimum eine schwierige Aufgabe sein [HS97]. Oft wird in dieser Situation auch ein bergiges Gelände, in welchem eine Person, welche nur mit dem Strahl einer Taschenlampe ausgerüstet ist, zur Erklärung herangezogen [TR17]. Diese Person kennt die Landschaft nicht, möchte aber den Fuß des Berges erreichen. Mit der Taschenlampe würde die Person den Boden ausleuchten, um der steilsten Neigung nach unten zu folgen. Sollte der Abstieg nicht weiter

möglich sein, kann es sein, dass die einen Tiefpunkt erreicht hat, dieser aber nicht der tiefste Punkt der gesamten Landschaft ist.

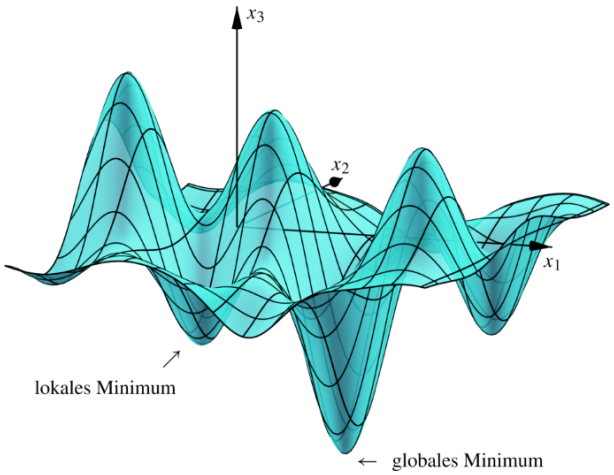


Abbildung 4: Lokales und globales Minimum

3.3.2 Befindet man sich wirklich im globalen Minimum?

Das Gradientenabstiegsverfahren finden in der Regel nur lokale Minima, abhängig vom gewählten Startpunkt [HS97]. Durch die fehlende Kenntnis der gesamten (komplexen) Funktion ist es nicht sichergestellt, dass das Verfahren das globale Minimum (bzw. das tiefte Tal im Beispiel 3.3.1) findet.

3.3.3 Wie löst man dieses Problem?

Es stehen eine Liste an Änderungen am Gradientabstiegsverfahren zur Verfügung.

- Initialisierung der Gewichte verändern:
  Man kann versuchen, die Initialisierung der Gewichte zu verändern, um den Lernerfolg zu verbessern. Dabei ist zu beachten, dass sowohl die Werte der Initialisierung für das Auffinden eines Minimums von Bedeutung ist, als auch der Startpunkt $x^0 \epsilon \mathbb{R}^n$ (3.2.1) des Gradientabstiegsverfahren, da dieser einen zentralen Einfluss darauf hat, welche Werte die Gewichte im Verlauf des Verfahrens annehmen.

  Zu beachten ist auch, dass die Initialisierung aller Gewichte auf denselben Zahlenwert dazu führt, dass die Gewichte in der Traininngsphase gleich verändert werden. Um diesem Problem entgegenzuwirken, wird die Initialisierung der Gewichte mit kleinen, um 0 herum streuenden Zufallsgewichten vorgenommen(symmetry breaking).
  Häufig kommt das sogenannte 'Multi-Start-Verfahren' zum Einsatz, bei dem die Berechnungen mit verschiedenen Startpunkten wiederholt werden.

- Lernparameter verändern:
  Neben Neu-Initialisierung der Gewichte kann der Lernparameter η (3.2.1) verändert werden. Das Erhöhen des Lernparameters hat größere Sprünge zum Minimum zur Folge. Vorteil dabei ist, dass flache Plateaus schneller durchlaufen werden. Beim Minimieren des Lernparameters ergibt sich der Vorteil, dass das globale Minimum nicht mehr so leicht übersprungen werden kann. Dabei wäre jedoch ein Nachteil, dass das Gradientenverfahren eine deutlich längere Laufzeit bekommt.
  Eine oft angewandte Kombination ist daher, eine stufenweise Veränderung der Lernrate im Verlauf des Gradientenabstieges [GR10, Seite 46].

Trotz der zahlreichen Lösungsmöglichkeiten, ist keine der Lösungen bei sämtlichen Problemen von Vorteil. Stattdessen ist oft simples ausprobieren notwendig, um die geeigneten Ansätze und Parameter auszuwählen. Ebenso können sich geeignete Methoden von Modell zu Modell unterscheiden [GR10, Seite 48].

4 Backpropagation

Inhalte der *Backpropagation*

Es wird ein Algorithmus gesucht, der die Wege ermittelt, welche einen stärkeren Einfluss auf den Output haben. Ebenso sollen diese Verbindungen dann gestärkt oder abgeschwächt werden. Dieses Kapitel erläutert, wie durch die Backpropagation ein neuronales Netz angepasst werden kann, um einen gewünschten Output zu erzielen.

- Was ist eine geeignete Verlustfunktion?
- Wie lernen Neuronale Netze?
- Grundidee Backpropagation
- Wie funktioniert der Backpropagation-Algorithmus

4.1 Was ist eine geeignete Verlustfunktion?

Eine weitverbreitete Verlustfunktion ist der sogenannte Mean Squared Error (MSE), bei dem der Durchschnitt der quadrierten Fehler berechnet wird. Bei Anwendung auf neuronale Netze lässt sich diese Funktion wie folgt ausdrücken:

$$C(w,b) = \frac{1}{2n} \sum_x \|\hat{y} - a\|^2. \tag{1}$$

In dieser Gleichung steht w für die Gesamtheit aller Gewichte im neuronalen Netz, während b alle Biases repräsentiert. n bezeichnet die Anzahl der Trainingsdatensätze und $\hat{y}$ ist der Vektor der Soll-Werte nach einem einzigen Trainingsbeispiel. Schließlich ist a der Vektor der tatsächlichen Ausgabewerte.

Wichtige Eigenschaften der MSE Verlustfunktion sind zum einen, dass sie niemals negativ wird, da jeder Term der Funktion positiv ist. Außerdem sieht man recht einfach, dass $C(w,b) \approx 0$ gilt, wenn die Soll-Werte ungefähr gleich den tatsächlichen Ausgabewerten sind.

Es wird der durchschnittliche Fehler über **alle** Trainingsbeispiele berechnet. Es gibt Methoden, bei denen der Fehler nur für ausgewählte Teilgruppen (sog. Batches) von Trainingsbeispielen berechnet wird (Batch-Methode). Diese Vorgehensweise kann die Laufzeit des Backpropagation-Algorithmus verbessern, wird in diesem Text jedoch nicht weiter vertieft.

4.2 Wie lernen Neuronale Netze?

Da wir nun eine Verlustfunktion kennen, welche den Fehler eines neuronalen Netzes bestimmt, kann man eben jene Verlustfunktion nutzen, um das neuronale Netz lernen zu lassen. Der Ausgabewert der Verlustfunktion hängt sowohl von allen Weights w als auch von allen Biases b des neuronalen Netzes ab. Nun sucht man die konkreten Werte für die Weights und Biases, damit die Verlustfunktion so klein wie möglich wird. Dies tut man mit dem bereits oben beschriebenen Gradientenverfahren. Man braucht also den negativen Gradientenvektor der Verlustfunktion. [MN19]

4.3 Grundidee Backpropagation

Backpropagation stellt einen effizienten Algorithmus zur Berechnung dieses komplexen Gradientenvektors dar, den wir benötigen, um den Fehler in einem neuronalen Netzwerk zu minimieren. Der Fehler an der Ausgabeschicht muss durch das gesamte Netz propagiert werden, um die Gewichte und Biaswerte entsprechend anzupassen. Für jedes Gewicht und jeden Biaswert möchten wir bestimmen, wie stark der Fehler von diesem abhängt. Hierbei werden die partiellen Ableitungen verwendet, die die Sensitivität einer Funktion in Bezug auf eine Änderung einer bestimmten Variable messen. Konkret suchen wir nach den Werten von $\frac{\partial C}{\partial w^l_{jk}}$ und $\frac{\partial C}{\partial b^l_j}$, die partiellen Ableitungen der Kostenfunktion (C) nach den Gewichten (w^l_{jk}) und Biaswerten (b^l_j). [MN19]

4.4 Wie funktioniert der Backpropagation-Algorithmus

Für die Erklärung des Backpropagation-Algorithmus nehmen wir folgendes neuronales Netz an:

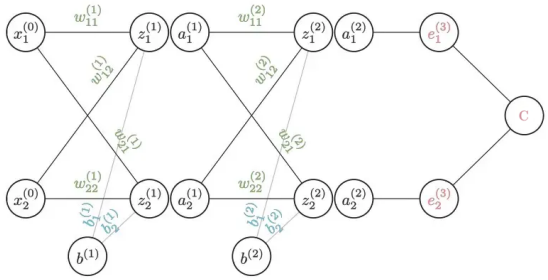


Abbildung 5: Quelle: https://towardsdatascience.com/understanding-backpropagation-abcc509ca9d0

Der gesamte Fehler C wurde in der Abbildung auf die zwei Output-Neuronen über $e_1^{(3)}$ und $e_2^{(3)}$ aufgeteilt. Außerdem wurde jedes Hidden-Neuron in seine gewichtete Summe z und seine Aktivierungsfunktion a aufgeteilt. Zuerst muss man sich überlegen, wie man die Weights / Biases, welche in die Output-Schicht einfließen, anpasst. Wenn man nun beispielsweise ausrechnen möchte, welchen Einfluss das Gewicht $w_{11}^{(2)}$ auf den Fehler C hat, dann muss man $\frac{\partial C}{\partial w_{11}^{(2)}}$ berechnen. Hierfür macht es Sinn, dass man sich anschaut wie das Weight $w_{11}^{(2)}$ in den Fehler C einfließt. Das Gewicht $w_{11}^{(2)}$ geht nur in den Teilfehler e_1 ein, daher muss man auch nur diesen betrachten.

$$
\begin{aligned}
e_1^{(3)} &= (\hat{y} - a_1^{(2)})^2 \\
a_1^{(2)} &= \sigma(z_1^{(2)}) \\
z_1^{(2)} &= w_{11}^{(2)} * a_1^{(1)} + w_{12}^{(2)} * a_2^{(1)} + b_1^{(2)} \\
\Rightarrow e_1^{(3)} &= (\hat{y} - \sigma(w_{11}^{(2)} * a_1^{(1)} + w_{12}^{(2)} * a_2^{(1)} + b_1^{(2)}))^2
\end{aligned}
$$

Möchte man nun die oben genannte partielle Ableitung ausrechnen, dann macht man sich die Kettenregeln zu Nutze und es gilt:

$$
\frac{\partial C}{\partial w_{11}^{(2)}} = \frac{\partial e_1^{(3)}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{11}^{(2)}}
$$

Diesen Schritt würde man für jedes Weight ausführen, welches in ein Output-Neuron einfließt. Anstatt jede partielle Ableitung einzeln aufzuschreiben, macht man sich das Hadamard-Produkt zu Nutze, um effizient alle partiellen Ableitungen dieser Weights zu

berechnen:

$$\begin{bmatrix} \frac{\partial C}{\partial w_{11}^{(2)}} & \frac{\partial C}{\partial w_{12}^{(2)}} \\ \frac{\partial C}{\partial w_{21}^{(2)}} & \frac{\partial C}{\partial w_{22}^{(2)}} \end{bmatrix} = \begin{bmatrix} \frac{\partial e_1^{(3)}}{\partial a_1^{(2)}} \\ \frac{\partial e_2^{(3)}}{\partial a_2^{(2)}} \end{bmatrix} \odot \begin{bmatrix} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \\ \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} \end{bmatrix} \odot \begin{bmatrix} \frac{\partial z_1^{(2)}}{\partial w_{11}^{(2)}} & \frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}} \\ \frac{\partial z_2^{(2)}}{\partial w_{21}^{(2)}} & \frac{\partial z_2^{(2)}}{\partial w_{22}^{(2)}} \end{bmatrix}$$

Die zwei linken Vektoren des Hadamard-Produkts werden im folgenden noch zu einem δ Term zusammengefasst, da diese in späteren Berechnungen öfters auftreten.

$$\delta^{(L)} = \frac{\partial C}{\partial A^{(2)}} \odot \frac{\partial A^{(2)}}{\partial Z^{(2)}}$$

$$\Rightarrow \frac{\partial C}{\partial W^{(2)}} = \delta^{(L)} \odot \frac{\partial Z^{(2)}}{\partial W^{(2)}}$$

Jetzt muss man sich noch anschauen, wie man die Abhängigkeit des Fehlers eines tiefer liegenden Weights / Biases berechnet, z.B. $\frac{\partial C}{\partial w_{11}^{(1)}}$. Die Grundidee bleibt dieselbe, jedoch können nun mehrere Pfade vom gesamten Fehler zu diesem Weight führen. In diesem Fall addieren wir die Kettenregel-Terme beider Pfade zusammen, bis sie sich an einem Neuron treffen.

$$\frac{\partial C}{\partial w_{11}^{(1)}} = \left(\frac{\partial e_1^{(3)}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial a_1^{(1)}} + \frac{\partial e_2^{(3)}}{\partial a_2^{(2)}} \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} \frac{\partial z_2^{(2)}}{\partial a_1^{(1)}} \right) \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial w_{11}^{(1)}}$$

In dieser Gleichung sieht man nun auch die sich wiederholenden δ^L Terme.

$$\frac{\partial C}{\partial w_{11}^{(1)}} = (\delta_1^L \frac{\partial z_1^{(2)}}{\partial a_1^{(1)}} + \delta_2^L \frac{\partial z_2^{(2)}}{\partial a_1^{(1)}}) \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial w_{11}^{(1)}}$$

Auch diese Berechnung führen wir für jedes "tiefere"Weight (in dem Fall von der 1. zur 2. Schicht) aus. Durch Matrizen lässt sich diese Berechnung wiefolgt ausdrücken:

$$\begin{bmatrix} \frac{\partial C}{\partial w_{11}^{(1)}} & \frac{\partial C}{\partial w_{12}^{(1)}} \\ \frac{\partial C}{\partial w_{21}^{(1)}} & \frac{\partial C}{\partial w_{22}^{(1)}} \end{bmatrix} = \begin{bmatrix} \delta_1^L \\ \delta_2^L \end{bmatrix}^T \cdot \begin{bmatrix} \frac{\partial z_1^{(2)}}{\partial a_1^{(1)}} & \frac{\partial z_1^{(2)}}{\partial a_2^{(1)}} \\ \frac{\partial z_2^{(2)}}{\partial a_1^{(1)}} & \frac{\partial z_2^{(2)}}{\partial a_2^{(1)}} \end{bmatrix} \odot \begin{bmatrix} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \\ \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \end{bmatrix} \odot \begin{bmatrix} \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}} & \frac{\partial z_1^{(1)}}{\partial w_{12}^{(1)}} \\ \frac{\partial z_2^{(1)}}{\partial w_{21}^{(1)}} & \frac{\partial z_2^{(1)}}{\partial w_{22}^{(1)}} \end{bmatrix}$$

$$\Rightarrow \frac{\partial C}{\partial W^{(1)}} = (\delta^L)^T \cdot \frac{\partial Z^{(2)}}{\partial A^{(1)}} \odot \frac{\partial A^{(1)}}{\partial Z^{(1)}} \odot \frac{\partial Z^{(1)}}{\partial W^{(1)}}$$

Die Berechnungen für die Bias-Werte sehen sehr ähnlich aus, daher werden diese hier nicht weiter ausgeführt. Somit kommt man auf die allgemeinen Formeln für die Backpropaga-

tion:

$$\frac{\partial C}{\partial W^{(l)}} = (\delta^{(l+1)})^T \cdot W^{(l+1)} \odot \frac{\partial A^{(l)}}{\partial Z^{(l)}} \odot X^{(l-1)}$$
$$\delta^{(L)} = \frac{\partial C}{\partial A^{(L)}} \odot \frac{\partial A^{(L)}}{\partial Z^{(L)}}$$
$$\frac{\partial C}{\partial B^l} = \delta^l$$

[MN19]

5 Trainieren und Testen

Inhalte von *Trainings- und Testdaten*

Im letzten Kapitel wird auf die Datensätze eingegangen, welche genutzt werden, um ein Modell mit Daten zu füllen und zu testen.

- Trainings- und Testdaten

5.1 Trainings- und Testdaten

Zum Trainieren eines Modells wird ein Datensatz benötigt. Dieser Datensatz wird in der Regel in mindestens drei verschiedene Datensätze unterteilt: Training-, Validierung- und Testdaten. Hier kann die Frage aufkommen, warum diese Unterteilung gemacht wird und nicht der vollständige Datensatz zum Lernen verwendet wird. Diese Frage sollte bei der Erklärung der Verwendungszwecke der jeweiligen Daten beantwortet werden.

5.1.1 Trainingsdaten

Ein Trainingsdatensatz ist ein Datensatz mit Beispielen. Konkret bedeutet das, dass hier eine Zuordnung zwischen Input -> Output bereits gegeben ist. Diese Daten werden genutzt, um die Gewichte (siehe 2.4.1) des neuronalen Netzes anzupassen [CA18, Seite 2ff]. Oft wird dieser Teildatensatz mit 70 bis 80 Prozent von der gesamten Datenmenge bemessen.

5.1.2 Validierungsdate

Der Validierungsdatensatz dient dazu, die Leistung des Modells während des Trainings zu bewerten und die Parameter (z.B. Lernrate oder Gewichte) zu optimieren. Dadurch wird das Modell verbessert [CA18, Seite 20]. Oft wird dieser Teildatensatz mit 10 Prozent von der gesamten Datenmenge bemessen.

5.1.3 Testdaten

Mithilfe des Testdatensatzes wird die Qualität des neuronalen Netzes nach dem Training überprüft. Wichtig ist dafür, dass dieser Datensatz zuvor nicht zum Lernen verwendet wurde, damit diese das Ergebnis nicht verfälschen. So kann festgestellt werden, wie das Modell auf Daten reagiert, welche nicht zuvor trainiert wurden [CA18, Seite 80f]. Oft wird dieser Teildatensatz mit 10 bis 15 Prozent von der gesamten Datenmenge bemessen.
Insgesamt sollte nun klar sein, warum die Datensätze aufgeteilt werden sollten. Wichtig anzumerken ist noch, dass die Verteilung der Datensätze variiert und so gewählt werden sollte, dass das Modell ausreichend gut trainiert und validiert wird und anschließend gründlich getestet wird.

5.1.4 Problem: Overfitting

Overfitting tritt auf, wenn das neuronale Netz zu stark an die Trainingsdaten angepasst ist und falsch auf neue, unbekannte Daten reagiert. Bei einer nur geringen Anzahl an Trainingsdaten werden nicht genug Anpassungen durchgeführt, sodass ein Lernen nach Mustern möglich wäre. Die Folge davon ist, dass die Trainingsdaten auswendig gelernt werden. Dadurch hat das neuronale Netz eine sehr hohe Genauigkeit bei den Trainingsdaten, jedoch eine sehr geringe bei dem Validierungsdatensatz [CA18, Seite 25].

6 Implementierung mit Keras

Die untenstehenden Codebeispiele orientieren sich am Leitfaden von Maximilian Wittman. [WM20]

6.1 Perceptron

Das folgende Perceptron ist ein neurales Netz, welches nur unterscheiden kann ob ein Bild eine 5 ist oder nicht. Die binäre Entscheidung wird mit Hilfe der Aktivierungsfunktion sigmoid getroffen.

```
1     from keras.datasets import mnist
2     from keras.models import Sequential
3     from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
4     from keras.utils import to_categorical
5     from PIL import Image
6     import numpy as np
7
8     # Laden Sie den MNIST-Datensatz herunter
9     (train_images, train_labels), (test_images, test_labels) =
          mnist.load_data()
10
```

```
11      # Wir muessen die Bilder neu formatieren, damit sie mit Keras
            funktionieren
12      train_images = train_images.reshape((60000, 28, 28, 1))
13      test_images = test_images.reshape((10000, 28, 28, 1))
14
15      # Normalisieren Sie die Bilder auf Werte zwischen 0 und 1
16      train_images, test_images = train_images / 255.0, test_images /
            255.0
17
18      # aendere Label zu 1, wenn 5 sonst 0
19      train_labels = np.where(train_labels == 5, 1, 0)
20      test_labels = np.where(test_labels == 5, 1, 0)
21
22      # Erstellen Sie das Modell
23      model = Sequential()
24      # Bild in Vektor umwandeln und Input Layer mit 28*28*1 Neuronen
            erstellen
25      model.add(Flatten(input_shape=(28, 28, 1)))
26      # Output Layer mit einem Neuron erstellen
27      model.add(Dense(1, activation='sigmoid'))
28      # Kompilieren Sie das Modell
29      model.compile(optimizer='adam',
30                    loss='binary_crossentropy',
31                    metrics=['accuracy'])
32      # Trainieren Sie das Modell
33      model.fit(train_images, train_labels, epochs=50, batch_size=64)
34      # Evaluieren Sie das Modell
35      test_loss, test_acc = model.evaluate(test_images, test_labels)
36      print('Test accuracy:', test_acc)
37      # Test accuracy konvergiert gegen 0.977
```

6.2 Neurales Netz mit hidden Layer

Das Ziel des neuralen Netzes aus dieser Implementation ist Ziffern von 0-9 aus Bildern zu identifizieren. Sie hat eine Input Layer, hidden Layer und eine Output Layer mit 10 Neuronen.

```
1       from keras.datasets import mnist
2       from keras.models import Sequential
3       from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
4       from keras.utils import to_categorical
5       from PIL import Image
6       import numpy as np
7
8       # Laden Sie den MNIST-Datensatz herunter
```

```
9     (train_images, train_labels), (test_images, test_labels) =
          mnist.load_data()
10
11    # Wir muessen die Bilder neu formatieren, damit sie mit Keras
          funktionieren
12    train_images = train_images.reshape((60000, 28, 28, 1))
13    test_images = test_images.reshape((10000, 28, 28, 1))
14
15    # Normalisieren Sie die Bilder auf Werte zwischen 0 und 1
16    train_images, test_images = train_images / 255.0, test_images /
          255.0
17
18    # Konvertieren Sie die Labels in kategorische Daten
19    train_labels = to_categorical(train_labels)
20    test_labels = to_categorical(test_labels)
21
22    # Erstellen Sie das Modell
23    model = Sequential()
24
25    # Fuegen Sie das erste Convolutional Layer hinzu
26    model.add(Flatten(input_shape=(28, 28, 1)))
27
28    # Fuegen Sie ein MaxPooling Layer hinzu
29    model.add(Dense(64, activation='relu'))
30
31    # Fuegen Sie das Output Layer hinzu. Da wir MNIST verwenden, haben
          wir 10 Nodes.
32    model.add(Dense(10, activation='softmax'))
33
34    # Kompilieren Sie das Modell
35    model.compile(optimizer='adam',
36                  loss='categorical_crossentropy',
37                  metrics=['accuracy'])
38
39    # Trainieren Sie das Modell
40    model.fit(train_images, train_labels, epochs=50, batch_size=64)
41
42    # Evaluieren Sie das Modell
43    test_loss, test_acc = model.evaluate(test_images, test_labels)
44    print('Test accuracy:', test_acc)
45    # Test accuracy konvergiert gegen 1.000
```

7 Quellenverzeichnis

7.1 Literatur

[CA18] Aggarwal, Charu C. (2018): Neural Networks and Deep Learning: A Textbook. Springer.

[TR17] Rashid, Tariq (2017): Neuronale Netze selbst programmieren. In O'Reilly eBooks. NY, USA

[GR10] Günter Daniel Rey und Karl F Wender (2010): Neuronale Netze, eine Einführung in die Grundlagen, Anwendung und Datenauswertung. 2. Auflage

[GBCL18] Goodfellow, I., Bengio, Y., Courville, A., & Lenz, G. (2018): "Deep Learning: das umfassende Handbuch : Grundlagen, aktuelle Verfahren und Algorithmen, neue Forschungsansätze."1. Auflage. Frechen: ISBN 978-3-95845-701-0.

7.2 Internetquellen

[HS97] Hochreiter, S. and Schmidhuber, J. (1997): Flat minima. Neural Computation, 9(1), 1-42. https://www.bioinf.jku.at/publications/older/3304.pdf (März 1996) (abgerufen am 11.05.2023)

[LH21] Linus Henning: Gradienten Verfahren zur Optimierung Neuronaler Netze. Weierstraß-Institut für Angewandte Analysis und Stochastik. https://www.wias-berlin.de/people/john/BETREUUNG/bachelor_henning.pdf (12.10.2021) (abgerufen am 12.05.2023)

[MN19] Determination Press, Michael Nielsen (2019): Neural Networks and Deep Learning

[WM20] Wittmann Maximilian (2020): Künstliche Intelligenz programmieren. Python programmieren. http://python-programmieren.maximilianwittmann.de/kunstliche-intelligenz-programmieren/ (abgerufen am 16.06.2023)

[BMDT16] Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., ... & Zieba, K. (2016). End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316. https://arxiv.org/pdf/1604.07316.pdf (abgerufen am 18.05.2023)

[VTY16] Vasilakos, A. V., Tang, Y., & Yao, Y. (2016). Neural networks for computer-aided diagnosis in medicine: a review. Neurocomputing, 216, 700-708. https://www.sciencedirect.com/science/article/pii/S0925231216308815 (abgerufen am 19.05.2023)

[YHCE18] Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., & Leskovec, J. (2018, July). Graph convolutional neural networks for web-scale recommender systems. In Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining (pp. 974-983). https://dl.acm.org/doi/pdf/10.1145/3219819.3219890 (abgerufen am 18.05.2023)

[WCQS18] Wang, T., Chen, Y., Qiao, M., & Snoussi, H. (2018). A fast and robust convolutional neural network-based defect detection model in product quality control. The International Journal of Advanced Manufacturing Technology, 94, 3465-3471. https://link.springer.com/article/10.1007/s00170-017-0882-0 (abgerufen am 20.05.2023)

[CJLV15] Chan, W., Jaitly, N., Le, Q. V., & Vinyals, O. (2015). Listen, attend and spell. arXiv preprint arXiv:1508.01211. https://arxiv.org/pdf/1508.01211.pdf (abgerufen am 27.05.2023)

[KSH17] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. Communications of the ACM, 60(6), 84-90. https://dl.acm.org/doi/pdf/10.1145/3065386 (abgerufen am 03.06.2023)