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 <Monat der Abgabe>

Zusammenfassung

Platz für das deutsche Abstract...

Abstract

Platz für das englische Abstract...

Inhaltsverzeichnis

Abbildungsverzeichnis

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. Der große Vorteil neuronaler Netze liegt darin, dass sie selbstständig lernen können, Muster in Daten zu erkennen, wodurch man die entsprechenden Algorithmen nicht mehr von Hand schreiben muss. Neuronale Netze sind dabei in der Lage, auch komplexe Muster und Zusammenhänge aus sehr großen Datenmengen zu erkennen und zu extrahieren.

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.

Häufig sind Muster zu komplex, um sie mit einem klassischen Algorithmus erfassen zu können oder sie sind schlecht von Hand mathematisch formulierbar. Ein Beispiel hierfür wäre die Erkennung von Tieren auf einem Foto. Die Eigenschaften, die z.B. eine Katze ausmachen von Hand in Bedingungen zu überführen ist nahezu unmöglich, gerade deswegen da die Katzen sehr verschieden aussehen können und aus verschiedenen Perspektiven auf dem Foto zu sehen sein können und die Menge an Merkmalen viel zu groß und komplex ist. 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.

Neuronale Netze sind häufig auch zuverlässiger bei der Lösung eines Problems, als klassische Algorithmen, da herkömmliche Algorithmen manchmal bestimmte Sonderfälle nicht abdecken können, wohingegen neuronale Netze anpassungsfähiger sind und teilweise mit solchen Sonderfällen umgehen können, da sie diese Fälle aus ihren Trainingsdaten extrahieren konnten, ein menschlicher Entwickler diese aber bei der Entwicklung eines Algorithmus übersehen hätte. Ein Beispiel für solche Fälle könnten autonom fahrende Fahrzeuge sein. Bei diesen ist es schwer möglich, Algorithmen zu schreiben, die unter allen Wetter-, Straßen- und Umweltbedingungen zuverlässig funktionieren. Schlecht zu sehende Straßenschilder oder Straßen mit undeutlichen Markierungen könnten ein Problem für solche Algorithmen darstellen, da handgeschriebene Algorithmen meistens auf genau solche Straßenmerkmale angewiesen sind. Neuronale Netze hingegen können möglicherweise mit solchen Fällen umgehen könnten, da sie sich aus den Trainingsdaten mehrere Faktoren abgeleitet haben, an denen sie sich orientieren können und zuverlässiger in der Erkennung von Mustern wie Straßenschildern sind.

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. Röntgenbilder können von neuronalen Netzen auf Anzeichen von Tumoren untersucht werden und können so medizinisches Fachpersonal bei Ihrer Arbeit unterstützen. 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. Dabei werden vor allem sogenannte “convolutional neural networks” verwendet. Dabei handelt es sich um eine Variante von neuronalen Netzen, die auf Grund der speziellen verwendeten Schichten und Aktivierungsfunktionen besonders gut für die Bilderkennung geeignet sind. Die Bilderkennung mittels neuronaler Netze findet auch in der Industrie Anwendung, um hergestellte Produkte automatisiert auf Mängel zu prüfen und auszusortieren.

Neuronale Netze können auch eingesetzt werden, um das Verhalten von Menschen zu analysieren. Anhand von Analytik Daten über die Interaktionen von Nutzern mit beispielsweise einer Website lassen sich mittels neuronaler Netze an den Nutzer angepasste Empfehlungen für Produkte, Videos oder ähnliches generieren. Die großen Mengen von anfallenden Daten über Nutzerinteraktionen können von solchen Netzen gefiltert und verarbeitet werden, um Muster im Verhalten der Nutzer zu finden, wodurch man auf die

Vorlieben des Anwenders Rückschlüsse ziehen kann.

Neuronale Netze sind grundsätzlich 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.

1.3 Grobes Prinzip

Neuronale Netze orientieren sich an der Funktionsweise von menschlichen Gehirnen und versuchen dadurch eine ähnliche Lernfähigkeit zu erzielen. Dabei werden Netze aus Neuronen simuliert, um menschliche Lernprozesse und kognitive Fähigkeiten nachzuahmen.

Die Neuronen neuronaler Netze sind in Schichten organisiert, die untereinander verbunden sind. Bei einem sogenannten "feed-forward" neuronalen Netz werden die Signale dabei von den Ausgängen der Neuronen der einen Schicht zu den Eingängen der Neuronen der nächsten Schicht geleitet.

Neuronen implementieren Funktionen, die die Eingabedaten verarbeiten und ein Ausgabesignal liefern, die sogenannten Aktivierungsfunktionen. Dabei werden zunächst die Eingabedaten zusammengefasst, meist indem sie aufsummiert werden und anschließend wird eine Aktivierungsfunktion auf das Ergebnis angewendet. Dadurch soll eine nichtlinearität mit eingebracht und die Werte auf einen bestimmten Wertebereich beschränkt werden.

Die Verbindungen zwischen den Neuronen sind von variabler Stärke und sind der primäre Faktor, der beim Lernprozess verändert wird.

Neuronale Netze werden trainiert, indem man sie auf einen Trainingsdatensatz anwendet und die Verknüpfungen der Neuronen anpasst. Der Trainingsdatensatz enthält dabei beispielhafte Eingabedaten und die dazugehörige gewünschte Ausgabe.

Basierend auf der Abweichung der Ergebnisse des neuronalen Netzes von denen der Trainingsdaten werden die Verknüpfungen der Neuronen dann automatisiert angepasst, wodurch ein Lerneffekt erzielt wird. Dazu wird das Gradientenverfahren eingesetzt, damit die Optimierung des Netzes möglichst schnell geschieht. Nachdem das Netz trainiert wurde, kann man es dann mit Daten speisen, die noch nicht in den Trainingsdaten enthalten waren und das Netz bildet dann eine entsprechende Ausgabe, basierend auf den Zusammenhängen, die es aus dem Trainingsdatensatz extrahiert hat.

2 Neuronen

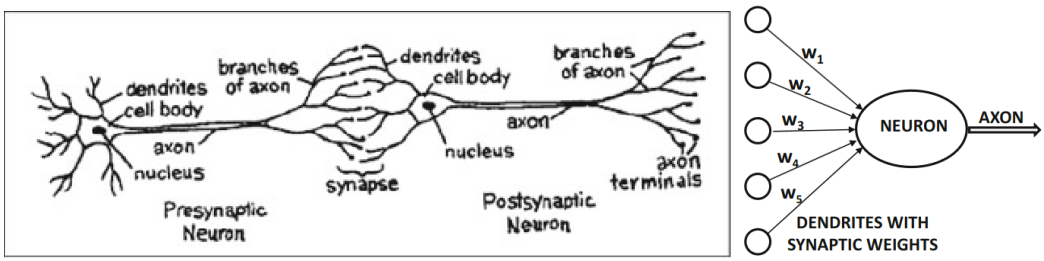
Inhalt

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.

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 Aktivierungsfunktions entscheidet, ob das Neuron in der nächsten Schicht aktiviert werden soll.

2.2.1.1 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 auf addiert. Das heißt, dass dieses Neuron das Ergebnis der Aktivierungsfunktion direkt beinflussen kann.

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.

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.

2.3 Funktionsweise

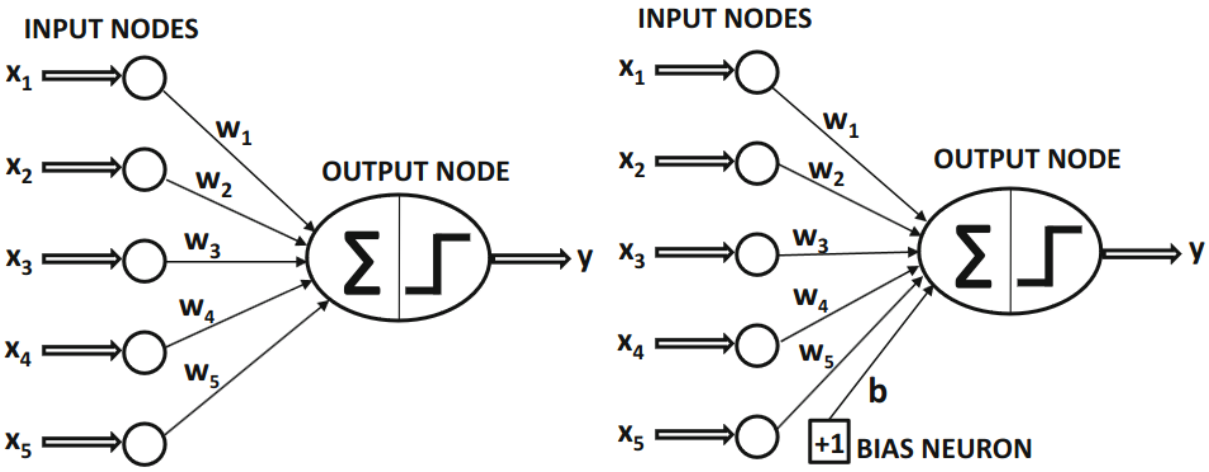
2.3.1 Perceptrons

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 $\hat{y}$.

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

In vielen Fällen wird ein nicht variables Element b mit in der Rechnung berücksichtigt. Dies verursacht das der Mittelwert der Vorhersage nicht 0 ist.

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



(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 der so genannten 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 entgegen zu wirken, 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 $\overline{X}$ einzelt 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. TODO Erklärung hier?

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.

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

2.3.2.1 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 reeler Wert ist.

$$\Phi(v) = v$$

2.3.2.2 Nicht lineare Aktivierung

In den frühen Tagen der Entwicklung von neuralen Netzen wurden sign, sigmoid und hyperbolic tangent Funktionen genutzt.

2.3.2.2.1 Sign Aktivierung

Die sign Funktion generiert nur binäre {-1,+1} Ausgaben. Aufgrund der Nichtstätigkeit der Funktion, können beim Trainieren keine Loss-Funktionen verwendet werden.

$$\Phi(v) = \text{sign}(v)$$

2.3.2.2.2 Sigmoid Aktivierung

Die Sigmoid Funktion generiert Werte zwischen 0 und 1. Sie eignet sich deshalb für Rechnungen die als Wahrscheinlichkeiten interpetiert werden sollen.

$$\Phi(v) = \frac{1}{1 + e^{-v}}$$

2.3.2.2.3 Tanh Aktivierung

Der Graph der Tanh Funktion hat eine ähneliche Form wie die der Sigmoid Funktion. Sie unterscheidet sich jedoch in der Skalierung, denn ihre Wertebereich liegt zwischen -1 und 1.

$$\Phi(v) = \frac{e^{2v} - 1}{e^{2v} + 1}$$

Die Tanh Funktion lässt sich auch durch die Sigmoid Funktion darstellen.

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

Wenn die Ausgabe der Berechnung postitiv 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.

2.3.2.3 Piecewise lineare Aktivierung

Historisch wurden die Sigmoid und Tanh Funktion zur Einführung von Nichtliniearität genutzt. Heutzutage sind piecewise linear activation Funktionen (Stückweise Linear) beliebter, weil diese das Trainieren von mehrschichtigen neuronalen Netzen einfacher machen.

2.3.2.3.1 ReLU

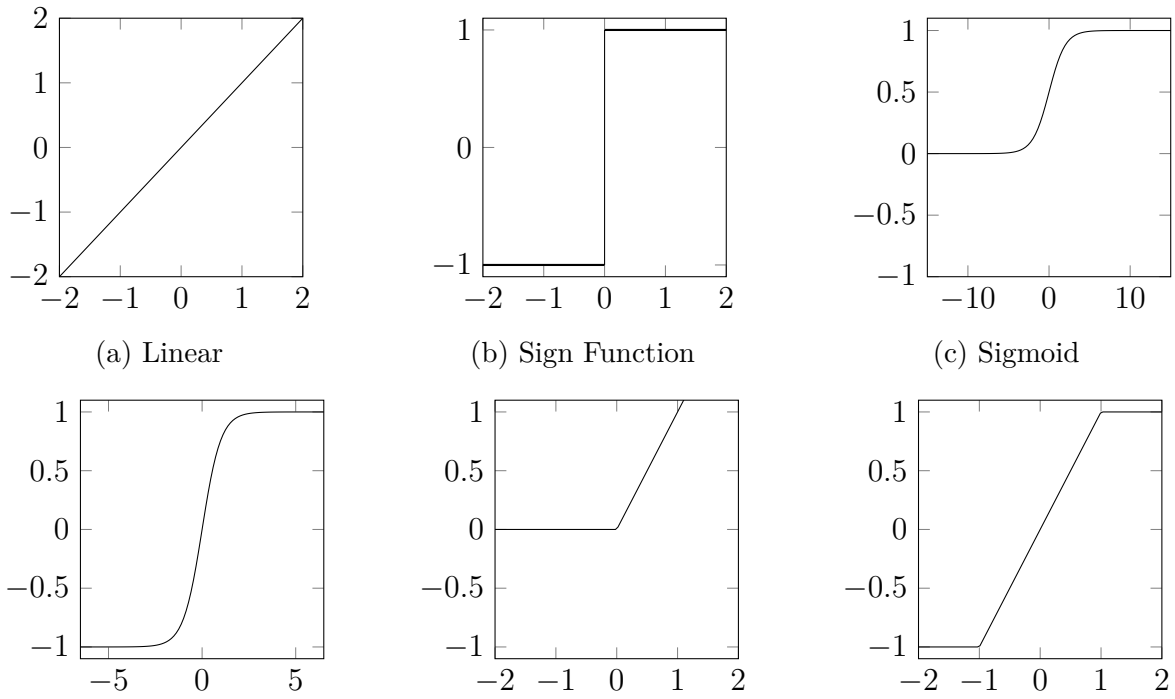
Das Ergebnis der ReLU (Rectified Linear Unit) Funktion ist 0, wenn die gewichtete Summe der Inputs v kleiner 0 ist, sonst ist das Ergbenis das unveränderte v.

$$\Phi(v) = \max\{v, 0\}$$

2.3.2.3.2 hard tanh Aktivierung

Das Ergbenis der hard tanh Funktion ist -1 für $v < 1$ oder 1 für $v > 1$. Sonst ist das Ergbenis das unveränderte v.

$$\Phi(v) = \max\{\min[v, 1], -1\}$$



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

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

Abbildung 3: Aktivierungsfunktionen

2.3.3 Loss Funktionen

2.3.4 Schichtenmodell

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 Werte der Neuronen, die die Ausgabeschicht bilden, sind auch die Werte, die als Ausgabe des Netzes gelten. Die Berechnungen, die in den Hidden-Layers stattfinden sind nicht nach außen hin sichtbar, lediglich die Ausgabe des Output-Layers gelangt nach außen.

Neuronale Netze sind in einer Struktur aus mehreren aufeinanderfolgenden Schichten aufgebaut. 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 von 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 mehrere Hidden-Layer verfügen und bei dem alle Schichten linear ohne Zyklen aufeinanderfolgen (”feed-forward”) und bei dem alle Neuronen einer Schicht mit allen Neuronen der folgenden Schicht verbunden sind. 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.

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.

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.

2.5 Fehler / Backpropagation Einführung

Nur eine sehr knappe Einführung, da eigenes Kapitel für dieses Thema reserviert

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 Gradientenabstiegsverfahren
- 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.

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 $x = 5, y = 3$

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

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 $y = 0$ rausbekommen. 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.

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.

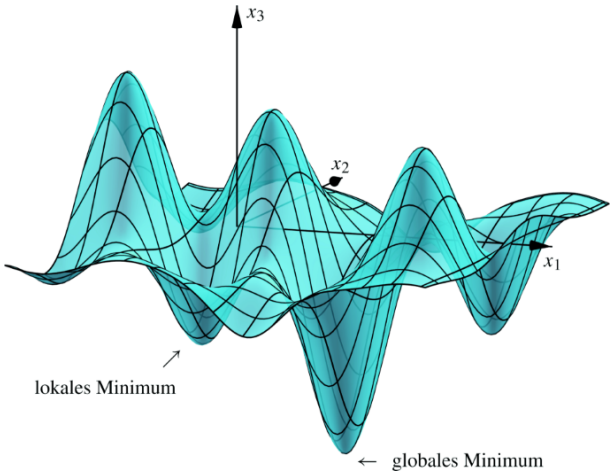


Abbildung 4: Lokales und globales Minimum

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].

- Impuls-Term hinzufügen:
  Der Impuls-Term multipliziert zum aktuellen Gradienten den vorangegangenen Gradienten. Dadurch wird zusätzlich zur Berechnung ein Anteil α des vorherigen Gradient berücksichtigt. Dadurch kann das Verfahren schneller gegen das Minimum konvergieren und hilft, die Wahrscheinlichkeit von Schritten in die falsche Richtung zu reduzieren.

  Mit der Funktion aus 3.2.1 wäre die angewandte Änderung des Verfahrens dann:

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

  Folgende Vorteile sind mit dem Impuls-Term verknüpft:

  - Lokale Minima werden eher übersprungen
  - Flache Plateaus werden aufgrund der Beschleunigung schneller durchlaufen.

  Jedoch birgt der Einsatz davon auch die erhöhte Gefahr des Überspringen des globalen Minimums [HS97].

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

Inhalt

Die Backpropagation umfasst folgende Elemente:

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

> Eine Einleitung muss auch durch die Arbeit führen. Sie muss dem Leser helfen, sich in der Arbeit und ihrer Struktur zu Recht zu finden. Für jedes Kapitel sollte eine ganz kurze Inhaltsangabe gemacht werden und ggf. motiviert werden, warum es geschrieben wurde. Oft denkt sich ein Autor etwas bei der Struktur seiner Arbeit, auch solche Beweggründe sind dem Leser zu erklären[a]:.

[a] [BBoJ], S. 6

4.1 Was ist eine geeignete Verlustfunktion?

Die Verlustfunktion wird genutzt um den Fehler eines neuronalen Netzes zu berechnen. Dieser Fehler gibt an, wie sehr die wirklichen Ausgabewerte des neuronalen Netzes nach Eingabe eines Trainingsdatensatzes von den gewollten Ausgabewerten abweichen.

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:

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

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 $y(x)$ ist der Vektor der Soll-Werte nach einzigen einem 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 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.

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).

4.4 Wie funktioniert der Backpropagation-Algorithmus

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

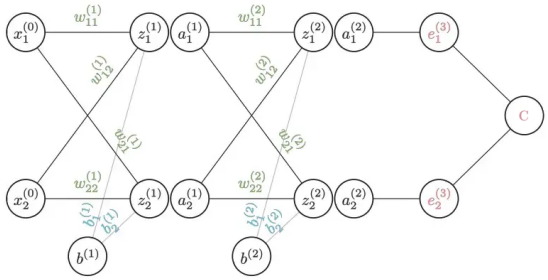


Abbildung 5: Hier Quelle angeben

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.

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

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 die selbe, 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.

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

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 Backpropagation:

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

5 Trainieren und Testen von Neuralen Netzen

todo

5.1 Trainingsdaten

todo

5.2 Testdaten

todo

6 Quellenverzeichnis

6.1 Literatur

[SW11] Stickel-Wolf, Christine; Wolf, Joachim (2011): Wissenschaftliches Lernen und Lerntechniken. Erfolgreich studieren—gewusst wie!. Wiesbaden: Gabler.

[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

6.2 Internetquellen

[BBoJ] Bertelsmeier, Birgit (o. J.): Tipps zum Schreiben einer Abschlussarbeit. Fachhochschule Köln-Campus Gummersbach, Institut für Informatik. http://lwibs01.gm.fh-koeln.de/blogs/bertelsmeier/files/2008/05/abschlussarbeitsbetreuung.pdf (29.10.2013).

[HR08] Halfmann, Marion; Rühmann, Hans (2008): Merkblatt zur Anfertigung von Projekt-, Bachelor-, Master- und Diplomarbeiten der Fakultät 10. Fachhochschule Köln-Campus Gummersbach.http://www.f10.fh-koeln.de/imperia/md/content/pdfs/studium/tipps/anleitungda270108.pdf (29.10.2013).

[JH20] Harrer, J. (2020): Künstliche Neuronale Netze.https://pxldeveloper.eu/assets/docs/KuenstlicheNeuronaleNetzeJulianHarrer.pdf (20.04.2023)

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

[HS97] Hochreiter, S. and Schmidhuber, J. (1997): Flat minima. Neural Computation, 9(1), 1-42.

[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)

A Anhang

A.1 Unterabschnitt von Anhang

TEXT FOLGT...

Erklärung über die selbständige Abfassung der Arbeit

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht.
Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

_____

(Ort, Datum, Unterschrift)

Hinweise zur obigen *Erklärung*

- Bitte verwenden Sie nur die Erklärung, die Ihnen Ihr **Prüfungsservice** vorgibt. Ansonsten könnte es passieren, dass Ihre Abschlussarbeit nicht angenommen wird. Fragen Sie im Zweifelsfalle bei Ihrem Prüfungsservice nach.
- Sie müssen **alle abzugebende Exemplare** Ihrer Abschlussarbeit unterzeichnen. Sonst wird die Abschlussarbeit nicht akzeptiert.
- Ein **Verstoß** gegen die unterzeichnete *Erklärung* kann u. a. die Aberkennung Ihres akademischen Titels zur Folge haben.