Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science
in Betriebswirtschaftslehre

an der Fakultät für Wirtschaftswissenschaften der
Universität Regensburg

Einführung in Machine Learning-Methoden

Eingereicht bei Herrn Prof. Dr. Daniel Rösch,
Lehrstuhl für Statistik und Risikomanagement
Betreut von Herrn Dr. Ralf Kellner

Eingereicht von Anna Menacher
Matr. Nr.: 1794800
Pröllerweg 6
94365 Parkstetten
anna.menacher@stud.uni-regensburg.de

Abgabetermin 08. Juni 2018

Zusammenfassung

Die vorliegende Arbeit stellt eine Einführung in die Thematik der künstlichen neuronalen Netze und somit in einen konkreten Teilbereich der Machine Learning-Methoden dar. Dabei sind künstliche neuronale Netze als nicht lineare, statistische Modelle aufzufassen, welche sowohl Regressions- als auch Klassifizierungsprobleme lösen und dementsprechend quantitative und qualitative Schätzungen abgeben können. Zu Beginn wird daher auf die Ermittlung der Schätzungen eingegangen, indem der Aufbau und die Funktionsweise von Schichten, sowie einzelner Neuronen, in einer Netzwerkarchitektur definiert werden. Die Parameter eines künstlichen neuronalen Netzes werden daraufhin mit dem Optimierungsverfahren Gradient Descent, welches eine Kostenfunktion minimiert, unter Verwenung von Backpropagation iterativ trainiert. Zuletzt sollen die theoretischen Erkenntnisse an dem sogenannten MNIST Datensatz angewandt werden. Das Ziel ist es hierbei verschiedene Hyperparameter des Modells zu testen und die Performance eines endgültigen Modells mit einer möglichst optimalen Kombination an Hyperparametern zu ermitteln. Das Gütemaß Accuracy erreicht dabei am finalen Modell eine Wert von 97,80%.

Inhaltsverzeichnis

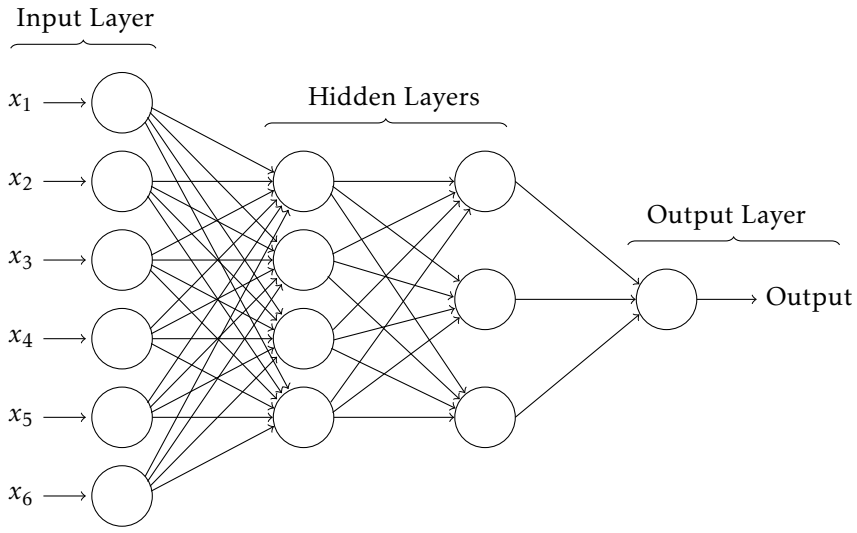
1 Künstliche neuronale Netze

Künstliche neuronale Netze eignen sich für die Lösung einer Vielzahl von Problemen. Insbesondere können sie jedoch Aufgabenstellungen des Supervised Learning bewältigen. Denn überwachtes Lernen umfasst sowohl Regressions- als auch Klassifizierungsprobleme, wobei es sich bei Regressionen um die Schätzung eines oder mehrerer quantitativer Outputs und bei Klassifizierungen um die Ermittlung eines oder mehrerer qualitativer Outputs handelt. Dies bedeutet, dass künstliche neuronale Netze sogar in der Lage sind Regressionsanalysen durchzuführen, welche gleichzeitig mehrere quantitative Ausgabewerte bestimmen. Selbiges gilt für Klassifizierungsprobleme, welche dann als Multi-Class Klassifizierungsproblem bezeichnet werden. Letztendlich beschreiben künstliche neuronale Netze aber lediglich nicht-lineare, statistische Modelle, welche die Fähigkeit besitzen komplexe Zusammenhänge darzustellen (vgl. Hastie et al., 2017, S.392).

1.1 Architektur eines künstlichen neuronalen Netzes

Allgemein setzt sich die Architektur eines künstlichen neuronalen Netzes aus drei Komponenten der Input Layer, den Hidden Layers und der Output Layer zusammen. Üblicherweise werden dabei in den jeweiligen Schichten die künstlichen Neuronen als Knoten und die Gewichte als Verbindungslinien in einem Graph veranschaulicht. Abbildung 1 stellt eine beispielhafte Netzwerkarchitektur mit drei Schichten dar, d.h. es handelt sich um ein künstliches neuronales Netz mit zwei Hidden Layers und einer Output Layer, wobei die Input Layer im Allgemeinen nicht zur Berechnung der Dichte eines Netzwerks hinzugezogen wird (vgl. Rashid, 2017, S.36 f., S.51 f.).

Abbildung 1 Architektur eines dreischichtigen neuronalen Netzes (vgl. Nielsen, 2015, Kapitel 1)



Die Neuronen der Input Layer sind für die Übertragung der Dateneingabe zur ersten Hidden Layer oder zur Output Layer verantwortlich. Prinzipiell besitzt die erste Schicht also keine tatsächliche Funktion in der Graphendarstellung eines neuronalen Netzes, da der jeweilige Input unverändert durch die Neuronen wieder ausgegeben wird. Nichtsdestotrotz

wird die Input Layer häufig der Übersicht halber zum Graph hinzugefügt.

Die mittleren Schichten eines künstlichen neuronalen Netzes ermöglichen einem Algorithmus die Inputs x möglichst korrekt auf die gewünschten Outputs y, sogenannte Targets bzw. Sollwerte, abzubilden. Da es sich bei künstlichen neuronalen Netzen zudem meistens um Methoden des Supervised Learning handelt, sind die Targets y zusätzlich zu den Eingabewerten x im Datensatz vorhanden. Die Outputs der Hidden Layers werden jedoch als versteckt bezeichnet, weil die Trainingsdatensätze keine Anhaltspunkte über die Targets der jeweiligen Hidden Neuronen enthalten. Zudem sind die Hidden Layers charakteristisch für Deep Learning, wobei ein Modell mindestens zwei Hidden Layers besitzen muss, um als Deep Neural Network bezeichnet zu werden.

Die letzte Schicht eines neuronalen Netzes ist die Output Layer. Die Anzahl der Neuronen in der Output Layer ist abhängig vom Kontext des jeweiligen Datensatzes. So besitzen Regressionsprobleme im Normalfall nur ein Output Neuron, während bei Klassifizierungsproblemen eine beliebige Anzahl an Neuronen die letzte Schicht im Netzwerk bilden kann. Ein künstliches neuronales Netz ist demzufolge in der Lage sowohl ein binäres als auch ein Multi-Class Klassifizierungsproblem zu lösen (vgl. Goodfellow et al., 2016, S.164 f.).

1.2 Künstliche Neuronen

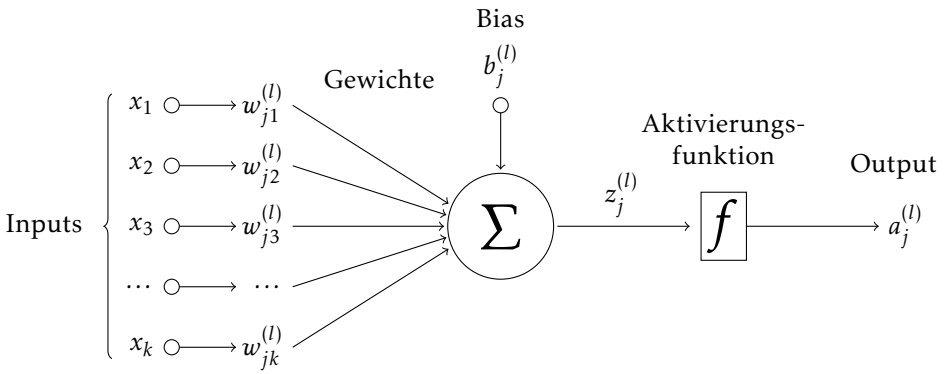
Neuronen sind für die Verarbeitung von Informationen innerhalb eines neuronalen Netzes verantwortlich. Sie setzen dabei die Input Signale x in Beziehung zu den Output Signalen y und sind somit die zentralen Bestandteile eines künstlichen neuronalen Netzes. Im Folgenden wird hauptsächlich auf die Funktionsweise von Hidden und Output Neuronen eingegangen, da die Input Neuronen keine weitere Aufgabe besitzen als ihre Eingabewerte unverändert wieder auszugeben (vgl. Lantz, 2013, S.208, S.212).

1.2.1 Aufbau eines künstlichen Neurons

Abbildung 2 beschreibt den Aufbau bzw. die Funktionsweise eines künstlichen Neurons, wobei die Inputs $x_1, \ldots, x_k$ einer Beobachtung i mehrere Schritte durchlaufen bevor sie das Neuron transformiert wieder verlassen. Nachfolgende Betrachtungen basieren somit auf einem Datenpunkt $(x_1^{(i)}, \ldots, x_k^{(i)}, y^{(i)})$ und nicht auf allen n Beobachtungen des Trainingsdatensatzes.

Im ersten Schritt werden die Eingabewerte x_k bzw. die Outputs der Neuronen der vorherigen Hidden Layer $a_k^{(l-1)}$ mit zu Beginn zufällig initialisierten Gewichten $w_{jk}^{(l)}$ multipliziert und anschließend aufsummiert. Die Gewichte sind dabei in der Lage bestimmte Input Signale zu verstärken oder auch abzuschwächen. Die Indizes k und j der jeweiligen Variablen beschreiben im Allgemeinen die Verbindung zwischen dem Input Signal k und dem Ouput Signal j. Das Superskript (l) steht hingegen für die Schicht, in welcher man sich befindet. Somit steht z.B. das Gewicht $w_{23}^{(4)}$ für die Verbindung zwischen dem 3. Neuron in der 3. Schicht und dem 2. Neuron in der 4. Schicht. Da die Outputs der Neuronen in einem Feedforward Neural Network von links nach rechts berechnet werden, handelt es sich bei

Abbildung 2 Funktionsweise eines künstlichen Neurons (vgl. Dering und Tucker, 2017)



Ermittlung der Outputs der ersten Hidden Layer um die Inputs x_k aus dem Trainingsdatensatz. Für die Berechnung aller weiteren Hidden oder Output Neuronen werden jedoch die Outputs der Neuronen der vorherigen Hidden Layer $a_k^{(l-1)}$ als Eingabewerte verwendet, d.h. der Output eines Neurons $a_j^{(l)}$ ist gleichzeitig der Input des nächsten Neurons $a_k^{(l-1)}$.

Im zweiten Schritt werden die resultierenden Linearkombinationen, bestehend aus den gewichteten Inputs, mit einem zusätzlichen Bias $b_j^{(l)}$ versehen, welcher in der Lage ist den Schwellenwert Θ einer Aktivierungsfunktion zu verändern. Die Linearkombination $\sum_k w_{jk}^{(l)} x_k$ inklusive Bias $b_j^{(l)}$ wird in der Variable $z_j^{(l)}$ zusammenfassend dargestellt (vgl. Bishop, 1995, S.117-119).

$$z_j^{(l)} \equiv \sum_k w_{jk}^{(l)} x_k + b_j^{(l)} \tag{1}$$

Im letzten Schritt zur Ermittlung des Outputs eines Neurons wird das Ergebnis der Summe $z_j^{(l)}$ durch eine nichtlineare Aktivierungsfunktion $f(\cdot)$ geschickt, welche letztlich den Output $a_j^{(l)}$ eines künstlichen Neurons berechnet (vgl. Haykin, 1999, S.32-34).

$$a_j^{(l)} = f\left(\sum_k w_{jk}^{(l)} x_k + b_j^{(l)}\right) = f\left(z_j^{(l)}\right) \tag{2}$$

Um jedoch in der Praxis obige Gleichungen effizient in Programmiersprachen implementieren zu können, müssen deren Variablen in Matrixschreibweise dargestellt werden. So werden z.B. die Variablen der Gleichung (2) folgendermaßen notiert:

$$\boldsymbol{a}^{(l)} = f\left(\boldsymbol{W}^{(l)}\boldsymbol{x} + \boldsymbol{b}^{(l)}\right) = f\left(\boldsymbol{z}^{(l)}\right) \tag{3}$$

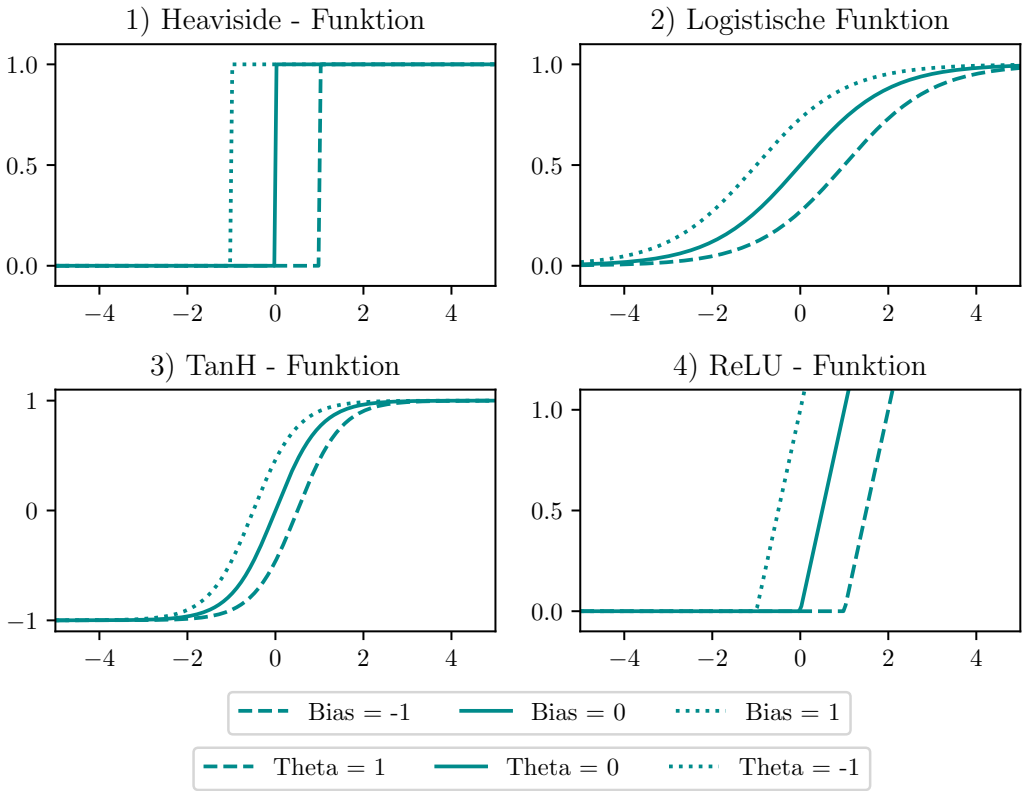
Die Matrix $\boldsymbol{W}^{(l)}$ beinhaltet sämtliche Gewichte zur Schicht (l), wobei die Zeilen j für die Gewichte in Verbindung mit den Neuronen der Schicht (l) und die Spalten k für die Gewichte in Verbindung mit den Neuronen der Schicht $(l-1)$ stehen. Die restlichen Variablen werden durch Spaltenvektoren dargestellt, d.h. $\boldsymbol{a}^{(l)}$ beinhaltet alle Outputs der Neuronen der Schicht (l), $\boldsymbol{x}$ umfasst alle Inputs einer Beobachtung des Trainingsdatensatzes, $\boldsymbol{b}^{(l)}$ beschreibt die Biaswerte der Schicht (l) und $\boldsymbol{z}^{(l)}$ fasst die gewichteten Inputs $\boldsymbol{W}^{(l)}\boldsymbol{x} + \boldsymbol{b}^{(l)}$ in einem Vektor zusammen (vgl. Nielsen, 2015, Kapitel 2).

1.2.2 Aktivierungsfunktionen

Aktivierungsfunktionen sind Funktionen, welche die Summe der gewichteten Inputs eines Neurons unter Beachtung eines Schwellenwerts Θ zu Outputs transformieren. Unter welchen Umständen ein Neuron als aktiviert bezeichnet wird, ist abhängig von der Größe des Outputs, sowie von den Eigenschaften der verwendeten Aktivierungsfunktion (vgl. Rashid, 2017, S.32-35). Ein weiterer Einflussfaktor auf die Aktivierung eines Neurons ist zudem der Schwellenwert, welcher dem negativen Bias innerhalb der Variable $z^{(l)}$ entspricht. Der Schwellenwert stellt somit die Grenze dar, die die Summe der gewichteten Eingabewerte zur Aktivierung des Neurons erreichen muss. Wenn es sich dabei um einen negativen Schwellenwert bzw. einen positiven Bias handelt, kann die Eingabeschwelle eines Neurons leichter erreicht werden. Ein positiver Schwellenwert bzw. ein negativer Bias verursacht einen gegenteiligen Effekt (vgl. Nielsen, 2015, Kapitel 1).

Im Folgenden werden nun die bekanntesten Aktivierungsfunktionen mit unterschiedlichen Bias Werten bzw. Schwellenwerten vorgestellt:

Abbildung 3 Aktivierungsfunktionen



1) Heaviside-Funktion

$$f(z) = H(z) = \begin{cases} 1, \text{ wenn } z \geq 0, \\ 0, \text{ sonst.} \end{cases} \tag{4}$$

Die Heaviside- oder Schwellenwert-Funktion beschreibt eine der einfachsten Aktivierungsfunktionen, wobei ein Neuron genau dann aktiviert wird, wenn die Summe

der gewichteten Inputs inklusive Bias größer oder gleich null ist. Dementsprechend bewirken positive Eingabewerte ein Output Signal von eins und negative Eingabewerte ein Output Signal von null. Jedoch sind der Schwellenwert-Funktion aufgrund ihrer sprungartigen Aktivierung eines Neurons andere Aktivierungsfunktionen vorzuziehen, welche in der Lage sind differenzierte Signale an nachfolgende Neuronen weiterzugeben (vgl. Rashid, 2017, S.32 f.).

2) Logistische Funktion

$$f(z) = \sigma(z) = \frac{1}{1 + \exp(-z)} \tag{5}$$

Die logistische oder Sigmoid-Funktion transformiert die gewichteten Inputs inklusive Bias auf einen Wertebereich zwischen null und eins. Im Gegensatz zur Heaviside-Funktion ist die logistische Funktion somit in der Lage schwache ($0 < \sigma(z) < 0,5$) oder starke Signale ($0,5 \leq \sigma(z) < 1$) an andere Neuronen in einem künstlichen neuronalen Netz weiterzugeben. Die Weitergabe von schwachen und starken Signalen ist an die Funktionsweise eines biologischen Neurons angelehnt, welches eine sogenannte *firing rate* besitzt. An dieser Stelle wird jedoch auf weitere biologische Analogien verzichtet, da in der Praxis häufig andere Aktivierungsfunktionen, wie z.B. die im Folgenden betrachtete Rectified Linear Unit-Funktion, verwendet werden, welche nicht auf der Funktionsweise eines biologischen Neurons aufbauen (vgl. Karpathy, 2017a).

3) Hyperbolische Tangens-Funktion

$$f(z) = tanh(z) = \frac{\exp(2z) - 1}{\exp(2z) + 1} = 2\sigma(2z) - 1 \tag{6}$$

Die hyperbolische Tangens- oder kurz TanH-Funktion gehört ebenfalls der Sigmoid-Familie an, wobei der Wertebereich des Tangens Hyperbolicus durch das Intervall $(-1,\ 1)$ definiert ist. Die Aktivierung eines künstlichen Neurons erfolgt jedoch ebenso wie bei der logistischen Funktion sukzessiv, mit dem Unterschied, dass die Eingabeschwelle bei $z = 0$ zu einem Output Signal von null führt. Ein schwaches Signal ist somit durch das Intervall $(-1,\ 0)$ definiert, wohingegen ein starkes Signal durch das Intervall $[0,\ 1)$ festgelegt wird (vgl. Géron, 2017, S.264).

4) Rectified Linear Unit-Funktion

$$f(z) = ReLU(z) = \max(0, z) \tag{7}$$

Die Rectified Linear Unit- oder kurz ReLU-Funktion besitzt einen Wertebereich, welcher durch das Intervall $[0,\ \infty)$ definiert ist. Bis die summierten Eingabewerte inklusive Bias den Schwellenwert $z = 0$ überschreiten, gilt ein Neuron als nicht aktiviert und besitzt somit ein Output Signal von null. Nach der Überschreitung der Eingabeschwelle saturiert die ReLU-Funktion nicht gegen einen maximalen Output Wert, sondern entspricht der Variable z (vgl. Géron, 2017, S.264).

Die oben genannten Aktivierungsfunktionen werden prinzipiell nur für die Neuronen der Hidden Layers verwendet, denn bei den Neuronen der Output Layer muss eine Fallunterscheidung zwischen Klassifizierungs- und Regressionsproblemen getroffen werden. So eignet sich die logistische Funktion insbesondere bei binären Klassifizierungsproblemen, da

durch die Begrenzung des Wertebereichs der Sigmoid-Funktion auf das Intervall $(0,\ 1)$ deren Outputs als Wahrscheinlichkeiten interpretiert werden können. Für Klassifizierungsprobleme mit mehr als zwei Klassen kann die logistische Funktion außerdem verallgemeinert werden. Diese Erweiterung wird als Softmax Funktion[1] bezeichnet.

$$\hat{y}_j = a_j^{(L)} = f(z_j^{(L)}) = softmax(z_j^{(L)}) = \frac{\exp(z_j^{(L)})}{\sum_k \exp(z_k^{(L)})} \tag{8}$$

Sei nun beispielsweise in einem künstlichen neuronalen Netz mit vier Output Neuronen bzw. Klassen $\hat{y}_j$ folgende vier gewichtete Inputs $z_j^{(L)}$ gegeben: $z_1^{(L)} = -1$, $z_2^{(L)} = 0$, $z_3^{(L)} = 1$ und $z_4^{(L)} = 2$. Setzt man die beliebig gewählten gewichteten Inputs als Eingabewerte in die Softmax Funktion ein, erhält man die Output Werte $\hat{y}_1 = 0,0321$, $\hat{y}_2 = 0,0871$, $\hat{y}_3 = 0,2369$ und $\hat{y}_4 = 0,6439$. Auf Basis der Outputs erkennt man, dass das Neuron 4 mit 64,39% die höchste Wahrscheinlichkeit besitzt und somit die Beobachtung der Klasse 4 zugeordnet wird (vgl. Nielsen, 2015, Kapitel 3).

Wie am obigen Beispiel zudem zu sehen ist, liegen alle Output Werte einer Softmax Funktion in einem Intervall zwischen null und eins und ergeben in Summe immer eins ($\sum_j^P \hat{y}_j = 0,0321 + 0,0871 + 0,2369 + 0,6439 = 1$). Aufgrund dieser beiden Eigenschaften kann man daher eine Wahrscheinlichkeitsverteilung über P Klassen angeben. Dabei ist die Wahrscheinlichkeit, dass das Ereignis j eintritt, für jede der P Klassen folgendermaßen definiert: $\hat{y}_j = P(y = j \mid z_j^{(L)})$ (vgl. Goodfellow et al., 2016, S.178-184). In Vektorschreibweise wird die Klassenzuordnung 4 von obigem Beispiel somit durch den Vektor $[0\ \ 0\ \ 0\ \ 1]^\top$ dargestellt, d.h. es handelt sich hierbei nicht mehr um die geschätzten Wahrscheinlichkeiten, sondern um die Schätzung einer Klassenzuordnung. Diese Zuordnung erfolgt durch die Anwendung des $\arg\max$ Operators auf die Softmax Funktion (vgl. Géron, 2017, S.141 f.).

Wenn es sich nicht um eine Klassifizierung, sondern um eine Regression einer metrischen abhängigen Variablen handelt, verwendet man lediglich ein Output Neuron mit einer linearen Aktivierungsfunktion, d.h. im Normalfall die Identitätsfunktion $f(z) = z$ (vgl. Bishop, 2009, S.228, S.236).

2 Training eines künstlichen neuronalen Netzes

Das Training eines Feedforward Neural Network erfolgt in zwei Schritten. Im ersten Schritt werden die Outputs des künstlichen neuronalen Netzes $\hat{\boldsymbol{y}}$ durch einen sogenannten Forward Pass bestimmt, d.h. die Inputs $\boldsymbol{x}$ durchlaufen die Schichten eines neuronalen Netzes bis die Output Neuronen $\hat{\boldsymbol{y}}$ als Schätzungen für $\boldsymbol{y}$ ausgeben. Im zweiten Schritt werden die Fehler zwischen den Targets $\boldsymbol{y}$ und den Schätzungen $\hat{\boldsymbol{y}}$ über eine Kostenfunktion ermittelt und im Anschluss auf die einzelnen Gewichte im Netzwerk zurückgeführt, um die Gewichte und die Biaswerte zu optimieren. Dieser Vorgang wird durch den Learning Algorithmus Gradient Descent unter zur Hilfenahme von Backpropagation ermöglicht (Lantz, 2013, S.215-217).

[1] Da es sich bei der Berechnung der Outputs der Neuronen um die letzte Schicht im Netzwerk handelt, wird der Superskript L und nicht l verwendet.

2.1 Forward Pass

In Kapitel 1.2.1 wird bereits die Funktionsweise eines einzelnen Neurons erklärt. In einem Forward Pass, auch häufig Forward Propagation genannt, durchlaufen die Inputs $\boldsymbol{x}$ hingegen eine Vielzahl an Neuronen bevor der letztendliche Output Vektor $\hat{\boldsymbol{y}}$ berechnet wird. Im Folgenden wird ein kompletter Forward Pass an einem beispielhaften künstlichen neuronalen Netz durchgeführt.

Sei also ein binäres Klassifizierungsproblem mit sechs Input-Variablen $\boldsymbol{x} = [x_1, \ldots, x_6]^\top = [1, 2, 3, 4, 5, 6]^\top$ für die Beobachtung i gegeben. Das künstliche neuronale Netz besteht dabei vergleichbar mit der Architektur aus Abbildung 1 aus zwei Hidden Layers und der Output Layer, wobei die erste Hidden Layer vier Neuronen, die zweite Hidden Layer drei Neuronen und die Output Layer ein Neuron besitzt. Für die Hidden Layers und die Output Layer soll zudem die logistische Funktion $\sigma(z)$ als Aktivierungsfunktion verwendet werden.

Um jedoch den Output des künstlichen neuronalen Netzes berechnen zu können, müssen die Gewichtsmatrizen $\boldsymbol{W}^{(2)}$, $\boldsymbol{W}^{(3)}$ und $\boldsymbol{W}^{(4)}$ mit zufälligen Gewichten initialisiert werden.

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} -1,5 & 1,9 & 0,7 & -0,2 & 0,4 & 0,0 \\ -0,2 & 0,5 & -0,7 & 0,8 & 0,1 & 1,3 \\ 0,2 & 1,0 & 1,9 & -0,5 & 1,8 & -1,1 \\ -0,9 & 0,2 & -1,2 & 1,2 & -1,3 & 0,8 \end{bmatrix} \qquad \boldsymbol{W}^{(3)} = \begin{bmatrix} -0,3 & -1,3 & 0,2 & 2,1 \\ -0,2 & 0,3 & 0,7 & -0,7 \\ 0,2 & 2,2 & -0,7 & -1,0 \end{bmatrix}$$

$$\boldsymbol{W}^{(4)} = \begin{bmatrix} 2,3 & 0,6 & 0,6 \end{bmatrix}$$

Die Berechnung des Forward Pass[2] entsprechend der Gleichung (3) sieht letztendlich folgendermaßen aus:

$$\begin{aligned} \boldsymbol{a}^{(2)} &= \sigma(\boldsymbol{W}^{(2)}\boldsymbol{x}) & \boldsymbol{a}^{(3)} &= \sigma(\boldsymbol{W}^{(3)}\boldsymbol{a}^{(2)}) & \hat{\boldsymbol{y}} &= \sigma(\boldsymbol{W}^{(4)}\boldsymbol{a}^{(3)}) \\ \boldsymbol{a}^{(2)} &= \sigma([5,6 \;\; 10,2 \;\; 8,3 \;\; -1,0]^\top) & \boldsymbol{a}^{(3)} &= \sigma([-0,8 \;\; 0,6 \;\; 1,4]^\top) & \hat{\boldsymbol{y}} &= \sigma([1,5]) \\ \boldsymbol{a}^{(2)} &= [1,0 \;\; 1,0 \;\; 1,0 \;\; 0,3]^\top & \boldsymbol{a}^{(3)} &= [0,3 \;\; 0,6 \;\; 0,8]^\top & \hat{\boldsymbol{y}} &= [0,8176] \end{aligned}$$

In den obigen Gleichungen erkennt man die schrittweise Berechnung der Outputs eines künstlichen neuronalen Netzes. So werden in allen Schichten zuerst die gewichteten Inputs $\boldsymbol{z}^{(l)}$ durch Matrixmultiplikation berechnet und im Anschluss durch die logistische Funktion geschickt. Dabei bilden die Outputs der vorherigen Schicht die Inputs für die nächste Schicht, d.h. der Output Vektor der ersten Hidden Layer $\boldsymbol{a}^{(2)}$ bildet den Input Vektor für die Berechnung des Output Vektors der zweiten Hidden Layer $\boldsymbol{a}^{(3)}$. Letztendlich erhält man einen Output von 81,76%, welcher den Schwellenwert der logistischen Funktion von 50% übersteigt. Somit wird das Output Neuron aktiviert und die binäre Klassenzuordnung fällt auf die Klasse 1 und nicht auf die Klasse 0.

[2] Um der Abbildung 1 im Kapitel 1.1 treu zu bleiben, wird an dieser Stelle auf die Verwendung eines Bias verzichtet.

2.2 Backward Pass

Nachdem der Forward Pass abgeschlossen ist bzw. die Outputs $\hat{y}_j$ für alle P Neuronen der Output Layer vorliegen, wird der sogenannte Backward Pass durchgeführt. Im Backward Pass werden die Gewichte und die Biaswerte, welche zu Beginn nur zufällig initialisiert wurden, aktualisiert. Dieses Update erfolgt iterativ über mehrere Epochen, d.h. die Kombination aus Forward und Backward Pass muss mehrmals durchgeführt werden, damit der angewandte Learning Algorithmus gegen eine möglichst optimale Lösung für die gesuchten Parameter - die Gewichte und die Biaswerte - konvergiert.

2.2.1 Kostenfunktionen

Bevor der Backward Pass durchgeführt werden kann, muss eine geeignete Kostenfunktion[3] $C(\hat{y}, y)$ ausgewählt und berechnet werden, welche die Abweichungen zwischen dem gewünschten Output y aus dem Trainingsdatensatz und dem durch das künstliche neuronale Netz berechneten Output $\hat{y}$ ermittelt. Die Auswahl einer geeigneten Kostenfunktion erfolgt ebenso wie bei den Aktivierungsfunktionen der Output Layer über eine Fallunterscheidung zwischen Klassifizierungs- und Regressionsproblemen.

Um also den Fehler zwischen y und $\hat{y}$ in einem binären Klassifizierungsproblem zu ermitteln, eignet sich insbesondere die Cross-Entropy Kostenfunktion, falls in der Output Layer die logistische Funktion als Aktivierungsfunktion verwendet wird. Dabei ist der Sollwert y entweder eins für die Klasse 1 oder null für die Klasse 0. Der geschätzte Output Wert $\hat{y}$ stellt hingegen eine Wahrscheinlichkeit dar, wodurch $\hat{y} \in [0,1]$ gilt. Zudem entspricht der Cross-Entropy Fehler der negativen Log-Likelihood einer Bernoulli Verteilung, d.h. es handelt sich um einen Maximum Likelihood Ansatz. Die Auswahl der Parameterschätzungen erfolgt somit über die Minimierung der negativen Log-Likelihood, was einer Maximierung der Log-Likelihood gleichkommt (vgl. Mitchell, 1997, S.118).

$$C(\hat{y}, y) = -\left[y \ \log(\hat{y}) + (1-y) \ \log(1-\hat{y})\right] \tag{9}$$

Für ein Klassifizierungsproblem mit P Klassen, bei welchem die Softmax Funktion in der Output Layer verwendet wird, eignet sich die Erweiterung der Cross-Entropy Kostenfunktion für ein Multi-Class Szenario. Diese entspricht der negativen Log-Likelihood einer Multinoulli Verteilung. Eine Multinoulli Verteilung stellt dabei lediglich eine diskrete Wahrscheinlichkeitsverteilung über P Klassen da (vgl. Goodfellow et al., 2016, S.177-184).

$$C(\hat{y}, y) = -\sum_{j=1}^{P} \mathbf{1}\{y = j\} \ \log(\hat{y}_j) = -\sum_{j=1}^{P} y_j \ \log(\hat{y}_j) \tag{10}$$

Liegt nun der Output Vektor $\hat{\boldsymbol{y}} = [0,0321 \ \ 0,0871 \ \ 0,2369 \ \ 0,6439]^\top$ aus dem Beispiel einer Multi-Class Klassifizierung in Kapitel 1.2.2 mit dem Target Vektor $\boldsymbol{y} = [0 \ \ 0 \ \ 0 \ \ 1]^\top$ vor, dann würde die Kostenfunktion erst einen Fehler von null annehmen, wenn die Wahrscheinlichkeiten der Klassen 1, 2 und 3 auf die Klasse 4 verteilt werden. So beträgt der Wert der Kostenfunktion vor einer Optimierung $C(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\left(1 \cdot \log(0,6439)\right) = -\left(-0,4402\right) = 0,4402$,

[3] Zu beachten ist, dass im Englischen mehrere gleichwertige Bezeichnungen für die Kostenfunktion üblich sind, wie z.B. *cost function*, *error function* und *loss function*.

obwohl die Klassenzuordnung prinzipiell korrekt ist. Dies liegt darin begründet, dass der Cross-Entropy Fehler im Allgemeinen versucht die gesamte Wahrscheinlichkeitsmasse auf die korrekten Klassen umzuverteilen. Somit ist eine richtige Klassenzuordnung allein nicht ausreichend (vgl. Géron, 2017, S.141-143).

Für ein Regression mit einer metrischen, abhängigen Variable eignet sich im Gegensatz zu Klassifizierungsproblemen der quadratische Fehler (vgl. Ng, 2017a).

$$C(\hat{y}, y) = \frac{1}{2}(y - \hat{y})^2 \tag{11}$$

2.2.2 Gradient Descent

Der Gradient Descent Algorithmus stellt ein Optimierungsverfahren dar, welches durch iterative Updates der Parameter in der Lage ist die Kostenfunktion zu minimieren. Das Ziel des Learning Algorithmus ist es somit möglichst optimale Werte für die Gewichte und die Biaswerte schrittweise zu ermitteln, indem die Kostenfunktion einen Wert nahe 0 annimmt (vgl. Géron, 2017, S.113).

Die Grundidee des Gradient Descent Verfahrens wird dabei durch die Gleichung (12) in Verbindung mit Gleichung (13) für einen einzelnen Parameter, nämlich ein Gewicht $w_{jk}^{(l)}$, eines künstlichen neuronalen Netzes veranschaulicht:

$$w_{jk}^{(l)} \leftarrow w_{jk}^{(l)} + \Delta w_{jk}^{(l)} \tag{12}$$

$$\Delta w_{jk}^{(l)} = -\eta \frac{\partial C}{\partial w_{jk}^{(l)}} \tag{13}$$

Betrachtet man die zwei Gleichungen separat, wird in Gleichung (12) die allgemeine Update Regel beschrieben, d.h. das Gewicht $w_{jk}^{(l)}$ soll um einen gewissen Betrag $\Delta w_{jk}^{(l)}$ verändert werden. In Gleichung (13) wird die Größe des Betrags festgelegt, welcher sich aus drei Komponenten zusammensetzt: einem negativen Vorzeichen, einer Lernrate η und einer partiellen Ableitung $\frac{\partial C}{\partial w_{jk}^{(l)}}$.

Da aber ein künstliches neuronales Netz zumeist aus mehreren tausend und nicht nur aus einzelnen Gewichten $w_{jk}^{(l)}$ bzw. Biaswerten $b_j^{(l)}$ besteht, werden die Update Regeln für eine Schicht (l) im Allgemeinen zusammenfassend in Matrixnotation dargestellt (vgl. Ng, 2017b, S.10):

$$\boldsymbol{W}^{(l)} = \boldsymbol{W}^{(l)} - \eta \frac{\partial C(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{W}^{(l)} - \eta \, \nabla_{\boldsymbol{W}^{(l)}} C(\boldsymbol{W}, \boldsymbol{b}) \tag{14}$$

$$\boldsymbol{b}^{(l)} = \boldsymbol{b}^{(l)} - \eta \frac{\partial C(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{b}^{(l)}} = \boldsymbol{b}^{(l)} - \eta \, \nabla_{\boldsymbol{b}^{(l)}} C(\boldsymbol{W}, \boldsymbol{b}) \tag{15}$$

Anzumerken ist, dass sämtlichen Notationen der Kostenfunktion als äquivalent anzusehen sind, d.h. $C(\hat{y}, y)$ entspricht $C(\boldsymbol{W}, \boldsymbol{b})$ oder auch kurz C. Des Weiteren handelt es sich bei η, um die Lernrate des Optimierungsverfahrens, welche als positiver Hyperparameter die Stärke der Änderungen der Gewichte bzw. der Biaswerte beeinflusst. Die Gradienten $\nabla_{\boldsymbol{W}^{(l)}} C(\boldsymbol{W}, \boldsymbol{b})$ bzw. $\nabla_{\boldsymbol{b}^{(l)}} C(\boldsymbol{W}, \boldsymbol{b})$ stellen letztendlich Matrizen bzw. Vektoren dar, welche die partiellen Ableitungen der Kostenfunktion bezüglich der einzelnen Gewichte bzw.

der Biaswerte der Schicht (l) beinhalten. Hier beschreiben Gradienten somit die Richtung, die die größte Zunahme der Kostenfunktion bewirkt. Damit die Änderung der Gewichte bzw. der Biaswerte keine Zunahme, sondern eine Abnahme der Kostenfunktion verursacht, muss ein negatives Vorzeichen vor die Gradienten gestellt werden. Hauptbestandteil in der Durchführung des Learning Algorithmus Gradient Descent sind somit die Gradienten $\nabla_{\boldsymbol{W}^{(l)}} C(\boldsymbol{W}, \boldsymbol{b})$ bzw. $\nabla_{\boldsymbol{b}^{(l)}} C(\boldsymbol{W}, \boldsymbol{b})$, welche über den sogenannten Backpropagation Algorithmus in Kapitel 2.2.3 ermittelt werden (vgl. Mitchell, 1997, S.89-91).

$$\nabla_{\boldsymbol{W}^{(l)}} C(\boldsymbol{W}, \boldsymbol{b}) = \begin{bmatrix} \frac{\partial C}{\partial w_{11}^{(l)}} & \cdots & \frac{\partial C}{\partial w_{1k}^{(l)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial w_{j1}^{(l)}} & \cdots & \frac{\partial C}{\partial w_{jk}^{(l)}} \end{bmatrix} \qquad \nabla_{\boldsymbol{b}^{(l)}} C(\boldsymbol{W}, \boldsymbol{b}) = \begin{bmatrix} \frac{\partial C}{\partial b_{1}^{(l)}} \\ \vdots \\ \frac{\partial C}{\partial b_{j}^{(l)}} \end{bmatrix}$$

Letztendlich existieren mehrere Formen des Gradient Descent Verfahrens, welche sich prinzipiell nur in der Berechnung der Kostenfunktion unterscheiden. So beschreiben die in Kapitel 2.2.1 aufgeführten Kostenfunktionen lediglich den Fehler einer Beobachtung aus dem Trainingsdatensatz $(x_1^{(i)}, x_2^{(i)}, \ldots, x_{n_x}^{(i)}, y^{(i)})$. Diese Formulierung einer Kostenfunktion definiert die Optimierung der Parameter durch den sogenannten Stochastic Gradient Descent Algorithmus. Dabei stellt das Stochastic Gradient Descent Verfahren eine Abweichung des Gradient Descent Algorithmus dar. Des Weiteren wird in der Praxis häufig das Mini-Batch Gradient Descent Verfahren verwendet, welches als Kompromiss zwischen dem Gradient Descent und dem Stochastic Gradient Descent Algorithmus anzusehen ist.

Formal wird bei dem Gradient Descent Algorithmus die gemittelte Summe der Kostenfunktionen aller Beobachtungen des gesamten Trainingsdatensatzes $\frac{1}{n}\sum_{i=1}^{n} C(\hat{y}^{(i)}, y^{(i)})$ minimiert, wohingegen beim Stochastic Gradient Descent Verfahren nur die Kostenfunktion einer einzelnen Beobachtung $C(\hat{y}^{(i)}, y^{(i)})$ betrachtet wird. Der Kompromiss in Form des Mini-Batch Gradient Descent Algorithmus minimiert ähnlich wie bei dem Gradient Descent Algorithmus eine gemittelte Summe der Kostenfunktionen. Allerdings wird die zu minimierenden Kostenfunktion folgendermaßen dargestellt: $\frac{1}{m}\sum_{i=1}^{m} C(\hat{y}^{(i)}, y^{(i)})$. Somit wird in dieser Art des Optimierungsverfahrens die Summe der Kostenfunktionen nicht durch die Anzahl aller Beobachtungen n geteilt, sondern durch die Anzahl der Iterationen m. Dabei werden die Iterationen durch die Anzahl der Batches definiert, welche zur Vervollständigung einer Epoche benötigt werden, d.h. $m = n/n_m$. Die Kostenfunktionen im Mini-Batch Gradient Descent Algorithmus werden für jeweils einen Batch der Größe n_m, welcher ein Sample aus dem Trainingsdatensatz darstellt, berechnet. Letztendlich werden sowohl beim Stochastic als auch beim Mini-Batch Gradient Descent Algorithmus die Beobachtungen zufällig aus dem Trainingsdatensatz ausgewählt, wobei es sich zumeist um Ziehen mit Zurücklegen handelt. Obwohl es sich also bei dem Stochastic und Mini-Batch Gradient Descent Verfahren lediglich um eine Approximation von dem Gradient Descent Algorithmus handelt, sind die ersten beiden Varianten meistens aufgrund des geringeren Rechenaufwands vorzuziehen (vgl. Ng, 2017b, S.10-14).

2.2.3 Backpropagation

Für das Training eines künstlichen neuronalen Netzes mit einer Variante von Gradient Descent[4] werden die partiellen Ableitungen einer Kostenfunktion bezüglich der Gewichte und der Biaswerte benötigt. Backpropagation stellt dabei eine effiziente Methode dar, um die für die Optimierung der Parameter notwendigen partiellen Ableitungen zu ermitteln.

Im Gegensatz zur Forward Propagation orientiert sich der Backpropagation Algorithmus jedoch nicht von der Input Layer zur Output Layer, sondern von der Output Layer zur Input Layer. Somit werden die ermittelten Fehler der Output Layer anteilig durch das Netzwerk zurückgeführt. Formal betrachtet man zuerst die Gewichte $w_{jk}^{(L)}$ zwischen der letzten Hidden Layer $(L-1)$ und der Output Layer (L) und im Anschluss die Verallgemeinerung für sämtliche Gewichte $w_{jk}^{(l)}$ aller davor liegenden Schichten (l).

Im Folgenden werden die allgemeinen Formeln des Backpropagation Algorithmus formuliert. Die Herleitung der Formeln erfolgt im Anhang A an einem Klassifizierungsproblem mit einer Cross-Entropy Kostenfunktion, sowie der logistischen Funktion als Aktivierungsfunktion in allen Hidden Layers und der Output Layer.

Allgemein verwendet der Backpropagation Algorithmus die Kettenregel, um die partiellen Ableitungen zu berechnen. Wie in Gleichung (16) zu sehen ist, beeinflusst das Gewicht $w_{jk}^{(L)}$ die Kostenfunktion $C(\hat{y}, y)$ lediglich über die gewichteten Summe $z_j^{(L)}$, welche wiederum einen Einfluss auf den ermittelten Output $\hat{y}_j$ besitzt und dieser sich letztendlich auf die Kostenfunktion C auswirkt. Ein Vorteil der Kettenregel ist somit, dass sich diese Ausdrücke separat darstellen und berechnen lassen.

$$\frac{\partial C(\hat{y}, y)}{\partial w_{jk}^{(L)}} = \frac{\partial C}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \tag{16}$$

Dabei wird die erste Anwendung der Kettenregel auf die partielle Ableitung der Kostenfunktion nach der gewichteten Summe $\partial C/\partial z_j^{(L)}$ durch die partielle Ableitung der Kostenfunktion nach dem ermittelten Output $\partial C/\partial \hat{y}_j$ und die anschließende partielle Ableitung des ermittelten Outputs nach der gewichteten Summe $\partial \hat{y}_j/\partial z_j^{(L)}$ in einem separaten Ausdruck, nämlich dem Fehlerterm $\delta_j^{(L)}$, zusammengefasst. Da die partielle Ableitung $\partial C/\partial \hat{y}_j$[5] von der jeweilig gewählten Kostenfunktion C abhängt, kann dieser Ausdruck nur in Verbindung mit einer konkreten Kostenfunktion ausgewertet werden. Ebenso benötigt die anschließende partielle Ableitung $\partial \hat{y}_j/\partial z_j^{(L)}$ in Gleichung (17) eine konkreten Aktivierungsfunktion zur Evaluierung. Daher sollte bei der Wahl einer Aktivierungsfunktion darauf

[4] In den folgenden Ausführungen des Backpropagation Algorithmus wird durchgängig Stochastic Gradient Descent als Optimierungsverfahren verwendet, sodass die Kostenfunktion nur für eine Beobachtung i berechnet werden muss. Dennoch ist auch eine Anwendung der in Kapitel 2.2.2 beschriebenen anderen Varianten der Optimierung denkbar.

[5] Formal müsste eine Summe über alle Output Neuronen zur partiellen Ableitung $\partial C/\partial \hat{y}_j$ hinzugefügt werden. Da aber sämtliche abgeleiteten Terme - mit Ausnahme der Ableitung nach $\hat{y}_j$ - null ergeben, wird auf diese Formalität verzichtet.

geachtet werden, dass es sich um eine differenzierbare Funktion handelt.

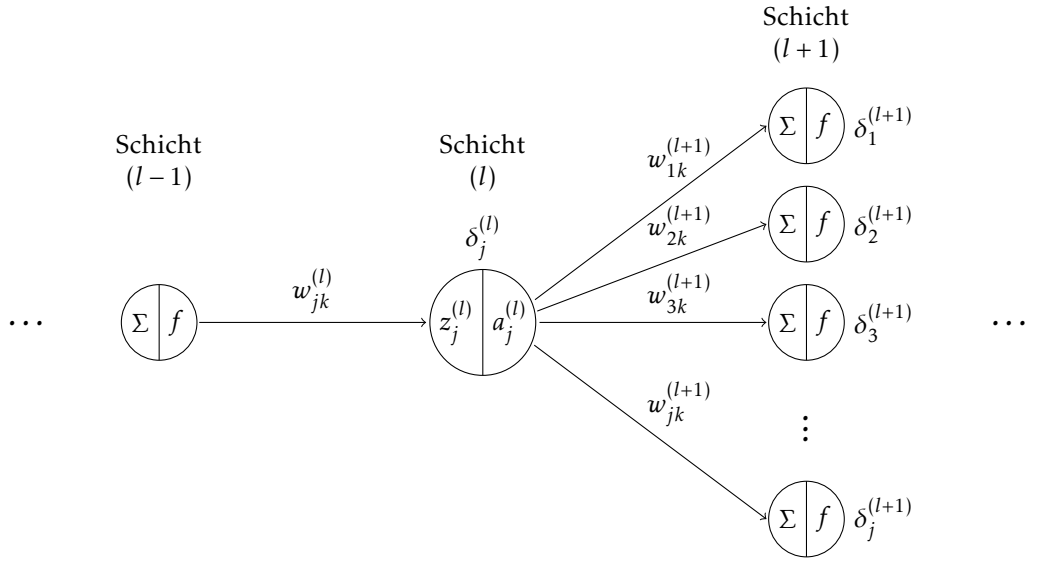
$$\delta_j^{(L)} \equiv \frac{\partial C(\hat{y}, y)}{\partial z_j^{(L)}} = \frac{\partial C}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial z_j^{(L)}} = \frac{\partial C}{\partial \hat{y}_j} f'(z_j^{(L)}) \tag{17}$$

Letztendlich ergibt sich durch das Einsetzten des Fehlerterms $\delta_j^{(L)}$ in Gleichung (16) und der Ableitung der gewichteten Summe $z_j^{(L)}$ aus Gleichung (1) nach dem Gewicht $w_{jk}^{(L)}$ folgende Gleichung:

$$\frac{\partial C(\hat{y}, y)}{\partial w_{jk}^{(L)}} = \delta_j^{(L)} a_k^{(L-1)} \tag{18}$$

Wie bereits in Kapitel 1.2.1 betont wurde, sollte man im Hinblick auf die Ableitung der gewichteten Summe nach einem Gewicht zwei notationsbedingten Sonderfälle beachten. So gilt für die Inputs $x_k = a_k^{(1)}$ und für die Outputs $\hat{y}_j = a_j^{(L)}$, d.h. allgemein entspricht $a_j^{(l)} = a_k^{(l-1)}$. Des Weiteren geht die Notation von der jeweiligen Schicht (l) aus und ändert sich bei der Betrachtung anderer Schichten im künstlichen neuronalen Netz entsprechend.

Abbildung 4 Funktionsweise von Backpropagation



Je weiter nun das Gewicht $w_{jk}^{(l)}$ von der Output Layer (L) entfernt ist, desto komplexer wird der Ausdruck zur Berechnung der partiellen Ableitung der Kostenfunktion bezüglich des Parameters. Um dies zu vermeiden wird der Fehlerterm $\delta_j^{(L)}$ für die Output Layer (L) in Gleichung (17) eingeführt, welcher in der Verallgemeinerung für jede Schicht (l) alle vorherigen Fehlerterme, die mit dem Gewicht verbunden sind, beinhaltet. Somit kann die Berechnung der partiellen Ableitung wie in Gleichung (18) dargestellt werden mit dem einzigen Unterschied, dass der Fehlerterm $\delta_j^{(l)}$ neu definiert werden muss.

$$\frac{\partial C(\hat{y}, y)}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)} \tag{19}$$

Abbildung 4 veranschaulicht diese neue Definition des Fehlerterms $\delta_j^{(l)}$, indem drei aufeinanderfolgende Schichten eines künstlichen neuronalen Netzes abgebildet werden. Weitere Neuronen der Schicht $(l-1)$ und (l) werden hierbei nicht aufgeführt, um den Fokus auf den Fehler $\delta_j^{(l)}$ in der Schicht (l) zu richten. Zudem geht die Notation von der Schicht (l) aus, weswegen das Neuron in dieser Schicht vergrößert dargestellt wird.

Um also den Fehlerterm für eine beliebige Schicht (l) zu ermitteln, muss man, wie bei Backpropagation üblich, die Abbildung von rechts nach links interpretieren, d.h. der neue Fehlerterm $\delta_j^{(l)}$ ergibt sich aus der gewichteten Summe der Fehlerterme $\delta_j^{(l+1)}$ aus der nachfolgenden Schicht $(l+1)$. Die gewichtete Summe setzt sich daher folgendermaßen zusammen: $\sum_j \delta_j^{(l+1)} w_{jk}^{(l+1)} = \delta_1^{(l+1)} w_{1k}^{(l+1)} + \cdots + \delta_j^{(l+1)} w_{jk}^{(l+1)}$. Durch diese neue Definition des Fehlerterms sind somit in den Fehlertermen der Schicht $(l+1)$ bereits sämtliche verbundene Fehlerterme der Schicht $(l+2)$ bis zur Output Layer (L) enthalten, wodurch sich die Notation trotz Anwendung der Kettenregel wesentlich kompakter darstellen lässt. Zum Schluss muss in der Schicht (l), ebenso wie bei der Berechnung des Fehlerterms in der Output Layer, die Aktivierungsfunktion nach $z_j^{(l)}$ abgeleitet werden (vgl. Bishop, 2009, S.242-245).

$$\delta_j^{(l)} \equiv \frac{\partial C(\hat{y}, y)}{\partial z_j^{(l)}} = \left(\sum_j \frac{\partial C}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial a_j^{(l)}} \right) \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \left(\sum_j \delta_j^{(l+1)} w_{jk}^{(l+1)} \right) f'(z_j^{(l)}) \tag{20}$$

Die Ermittlung der partiellen Ableitungen der Kostenfunktion nach den Biaswerten unterscheidet sich von der Berechnung der partiellen Ableitungen der Kostenfunktion nach den Gewichten nur dadurch, dass die Ableitung der gewichteten Summe $z_j^{(l)}$ nach einem beliebigen Biaswert $b_j^{(l)}$ immer eins und nicht $a_k^{(l-1)}$ oder x_k entspricht. Dadurch lässt sich die Formel für die partielle Ableitung durch den Fehlerterm $\delta_j^{(l)}$ darstellen (vgl. Mitchell, 1997, S.97-103).

$$\frac{\partial C(\hat{y}, y)}{\partial b_j^{(l)}} = \delta_j^{(l)} \tag{21}$$

Die effizientere Darstellung der obigen Gleichungen in Matrixnotation, d.h. die Berechnung der Gradienten für die Gewichtsaktualisierungen aus Gleichung (14) und (15), erfolgt im Anhang B.

3 Analyse des MNIST Datensatzes

Die Analyse des MNIST Datensatzes erfolgt in mehreren Schritten. Zu Beginn werden die Daten aufbereitet, sodass ein künstliches neuronales Netz diese als Inputs bzw. Outputs versteht. Im nächsten Schritt werden daraufhin die zu optimierenden Parameter - die Gewichte und die Biaswerte - initialisiert. Zuletzt werden zahlreicher Hyperparameter, d.h. freier Parameter des Modells, festgelegt, welche sowohl Auswirkungen auf die Performance als auch auf die Berechnungsdauer eines künstlichen neuronalen Netzes besitzen. Zuletzt wird die Performance des Modells - nach Bestimmung der Hyperparameter - an einem sogenannten Testdatensatz evaluiert.

3.1 MNIST Datensatz

Der MNIST Datensatz besteht aus insgesamt 70.000 handgeschriebenen Zahlen von High School Schülern und Angestellten des statistischen Bundesamts der USA, wobei jedes Bild mit einem Label, d.h. der korrekt abgebildeten Zahl, versehen ist. In Abbildung 5 sind 100 beispielhafte Bilder dieser Ziffern zwischen 0 und 9 aus dem Datensatz dargestellt, wobei alle Bilder eine Größe von 28 x 28 Pixeln besitzen. Aus diesen 28 x 28 Pixeln ergeben sich die 784 Input Variablen, welche die Farbwerte einer Graustufenpalette zwischen 0 und 255 annehmen können. Dabei steht 0 für einen weißen und 255 für einen schwarzen Farbwert (vgl. Rashid, 2017, S.133-141).

Abbildung 5 Beispielhafte Zahlen des MNIST Datensatzes

Durch den hohen Bekanntheitsgrad des MNIST Datensatzes ist dieser zudem bereits in vielen Python Libraries, wie z.B. TensorFlow oder Scikit-Learn, enthalten. Jedoch ist anzumerken, dass beispielsweise die Inputs des MNIST Datensatzes in der TensorFlow Library im Vorfeld aufbereitet wurden. Der Vollständigkeit halber wird daher in dieser Arbeit der ursprüngliche, unbearbeitete MNIST Datensatz aus dem Paper von Lecun et al. (1998) verwendet, welcher die Daten in einen Trainingsdatensatz mit 60.000 Beobachtungen und in einen Testdatensatz mit 10.000 Beobachtungen aufteilt. Um jedoch im Anschluss an die Trainingsphase die Hyperparameter eines künstlichen neuronalen Netzes zu optimieren, wird ein Teil des ursprünglichen Trainingsdatensatzes für einen sogenannten Validierungsdatensatz verwendet. Somit setzt sich der MNIST Datensatz in dieser Arbeit folgendermaßen zusammen:

Tabelle 1 Zusammensetzung des MNIST Datensatzes

Datensatz	Beobachtungen
Training	50.000
Validierung	10.000
Test	10.000

3.2 Aufbereitung der Eingabe- und Ausgabewerte

Der MNIST Datensatz enthält insgesamt 784 Input Variablen für jede Beobachtung i, wobei alle Eingabe- bzw. Pixelwerte $x_1, \ldots, x_{784}$ einen Wert zwischen 0 und 255 annehmen. Indem der Wertebereich aller Input Variablen also bereits durch das Intervall $[0, 255]$ definiert ist, ist eine Skalierung bzw. Standardisierung der Eingabewerte nicht zwingend notwendig. Nichtsdestotrotz hat es sich in der Praxis etabliert diese Maßnahmen trotzdem durchzuführen, da die Möglichkeit besteht, dass der Learning Algorithmus Gradient Descent mit standardisierten oder skalierten Inputs schneller gegen ein Minimum konvergiert. Im Extremfall hingegen erreicht ein Optimierungsverfahren keine Konvergenz gegen ein globales oder lokales Optimum, wenn die Input Variablen in unterschiedlichen Größenordnungen vorliegen und vor dem Training eines künstlichen neuronalen Netzes nicht auf ein einheitliches Skalenniveau gebracht werden (vgl. Bishop, 1995, S.298-300).

In Anlehnung an die Aufarbeitung des MNIST Datensatzes in der Python Library TensorFlow werden die Eingabewerte lediglich um den Faktor 1/255 skaliert, d.h. sie werden durch den maximalen Wert aus dem ursprünglichen Wertebereich $[0, 255]$ geteilt. Durch die lineare Transformation wird der Wertebereich der Input Variablen somit auf das Intervall $[0, 1]$ verschoben (vgl. Abadi et al., 2016).

Je nach Aufgabenstellung müssen zudem die Labels y an die Darstellung der berechneten Outputs $\hat{y}$ angepasst werden. Das Ziel der Analyse des MNIST Datensatzes ist es handgeschriebene Ziffern von 0 bis 9 zu erkennen, d.h. es handelt sich hierbei um ein Multi-Class Klassifizierungsproblem mit zehn Klassen. Somit enthält die Output Layer des künstlichen neuronalen Netzes zehn Output Neuronen, welche die Klassen 0 bis 9 repräsentieren. Die Anpassung der Klassen an die berechneten Outputs erfolgt durch die Umwandlung der Labels in einen Vektor der Länge zehn, wobei die Position der eins innerhalb des Vektors die Klassenzuordnung angibt. Diese sogenannten One-Hot Vektoren werden mit den entsprechenden Labels des MNIST Datensatzes in Tabelle 2 dargestellt (vgl. Mitchell, 1997, S.114 f.).

Tabelle 2 Transformation der Output Labels zu One-Hot Vektoren

Label	One-Hot Vektoren
0	[1 0 0 0 0 0 0 0 0 0]
1	[0 1 0 0 0 0 0 0 0 0]
2	[0 0 1 0 0 0 0 0 0 0]
3	[0 0 0 1 0 0 0 0 0 0]
4	[0 0 0 0 1 0 0 0 0 0]
5	[0 0 0 0 0 1 0 0 0 0]
6	[0 0 0 0 0 0 1 0 0 0]
7	[0 0 0 0 0 0 0 1 0 0]
8	[0 0 0 0 0 0 0 0 1 0]
9	[0 0 0 0 0 0 0 0 0 1]

Schlussendlich besteht die Aufbereitung der Eingabewerte des MNIST Datensatzes somit aus der Skalierung der Datenmatrizen des Trainings-, Validierungs- und Testdatensatzes um den Faktor 1/255. Dabei besitzt beispielsweise die Datenmatrix des Trainingsdatensat-

zes die Größe ($n \times n_x$) bzw. konkret (50.000 $\times$ 784). Die Anzahl der Beobachtungen und dadurch die Dimension der Matrizen der anderen Datensätze ist Tabelle 1 zu entnehmen.

Die Aufbereitung der Ausgabewerte des MNIST Datensatzes erfolgt hingegen durch die Transformation der Labels in One-Hot Vektoren. Die gewünschten Outputs werden somit durch die Matrix der Größe ($n \times P$) bzw. konkret (50.000 $\times$ 10) im Falle des Trainingsdatensatzes angegeben. Die Dimensionen der Matrizen für den Validierungs- und für den Testdatensatz ergeben sich ebenfalls aus Tabelle 1.

3.3 Initialisierung der Gewichte und Biaswerte

Bei den Gewichten und den Biaswerten handelt es sich um unbekannte Parameter, welche vor dem Training eines künstlichen neuronalen Netzes initialisiert werden müssen. Je nach Wahl der Startwerte können diese einen großen Einfluss auf die Funktionalität bzw. auf die Konvergenz eines Optimierungsverfahrens besitzen, weshalb die Methode der Initialisierung prinzipiell auch als Hyperparameter des Modells angesehen werden kann.

Innerhalb dieser Arbeit erfolgt die Gewichtsinitialisierung, indem die Werte einer Normalverteilung mit einem Mittelwert 0 und einer Standardabweichung 0,05 entnommen werden. Die zufällig gezogenen Startwerte liegen somit nahe null, wodurch der Learning Algorithmus, ähnlich wie bei der Skalierung der Inputs auf einen Wertebereich zwischen null und eins, möglicherweise schneller gegen ein Optimum konvergiert. Anzumerken ist zudem, dass die Gewichte nicht mit Nullen initialisiert werden dürfen, da sonst die partiellen Ableitungen ebenfalls null ergeben. Somit würde der Learning Algorithmus exakt gleiche Updates an den Gewichten durchführen und die Gewichte würden nach der Aktualisierung weiterhin unverändert vorliegen. Dies bezeichnet man häufig als Symmetrieproblem (vgl. Hastie et al., 2017, S.397 f.).

Die Biaswerte werden jedoch im Gegensatz zu den Gewichten in dieser Arbeit mit Nullen initialisiert, da diese kein Symmetrieproblem innerhalb des Netzwerks verursachen (vgl. Géron, 2017, S.268 f.).

3.4 Hyperparameter

Künstliche neuronale Netze bestehen aus einer Vielzahl von Hyperparametern, welche nach Aufbereitung der Daten und vor Beginn des Trainings eines Modells festgelegt werden müssen. Ein Set aus Hyperparametern besteht dabei beispielsweise aus folgenden Parametern: Anzahl der Hidden Layers, Anzahl der Neuronen, Aktivierungsfunktionen, Optimierungsverfahren, Kostenfunktionen, Lernraten, Batch Größen und Anzahl der Epochen. Aufgrund der hohen Anzahl von möglichen Kombinationen an Hyperparametern in einem künstlichen neuronalen Netz werden daher häufig automatisierte Verfahren, wie z.B. Grid Search, Random Search oder auch bayesianische Optimierungsverfahren, verwendet, um ein Modell mit geeigneter Performance zu finden. Obwohl in der Praxis diese automatisierten Verfahren einer manuellen Hyperparametersuche vorzuziehen sind, werden in dieser Arbeit die freien Parameter per Hand eingestellt. Dadurch sollen die Auswirkun-

gen unterschiedlicher Hyperparameter auf ein Modell aufgezeigt werden (vgl. Karpathy, 2017b).

Insbesondere wird jedoch im Folgenden auf die Anzahl der Hidden Layers, die Anzahl der Neuronen, die Aktivierungsfunktionen und die Lernrate eines Modells eingegangen. Die restlichen Hyperparameter werden entsprechend den Empfehlungen aus dem Paper von Bengio (2012) festgesetzt, d.h. als Optimierungsverfahren wird der Mini-Batch Gradient Descent Algorithmus mit einer Batch Size von 100 verwendet, welcher den Cross-Entropy Fehler minimiert. Bedingt durch die Verwendung des Cross-Entropy Fehler als Kostenfunktion, wird zudem die Softmax Funktion als Aktivierungsfunktion in der Output Layer festgelegt. Letztendlich werden die einzustellenden Hyperparameter konstant über 100 Epochen evaluiert.

3.4.1 Performancevergleich am Validierungsdatensatz

Der in Kapitel 3.1 vorgestellte Validierungsdatensatz eignet sich, um eine optimale Zusammensetzung von Hyperparameter innerhalb eines Netzwerks zu finden. Denn es können mehrere Modelle, welche auf Basis des Trainingsdatensatzes erstellt wurden, über die Performance am Validierungsdatensatz miteinander verglichen werden. Jedoch kommt es bei künstlichen neuronalen Netzen häufig zu einem Problem, welches als *overfitting* bezeichnet wird. Dabei handelt es sich um die zu starke Anpassung der Gewichte und der Biaswerte eines künstlichen neuronalen Netzes an die Beobachtungen des Trainingsdatensatzes. Nichtsdestotrotz kann das Risiko von *overfitting* verringert und die Fähigkeit eines Modells korrekte Schätzungen für neue Datenpunkte abzugeben gesteigert werden, wenn für die Hyperparameterwahl die Performance eines Modells an einem Validierungsdatensatz und nicht am Trainingsdatensatz gemessen wird. Letztendlich wird dann die Güte des finalen Modells mit einem möglichst optimalen Set an Hyperparametern an einem Testdatensatz evaluiert (vgl. Bishop, 2009, S.32).

Der Performancevergleich erfolgt dabei aufgrund der gleichmäßigen Klassenverteilung der Ziffern innerhalb der Datensätze aus Tabelle 1 durch das Gütemaß Accuracy,[6] welches den Anteil der korrekt getroffenen Klassenzuordnungen beschreibt. Dieser Anteil wird ermittelt, indem nur die Beobachtungen i aufsummiert werden bei denen die One-Hot Vektoren des geschätzten Outputs $\hat{y}$ und des gewünschten Outputs y einer Beobachtung genau übereinstimmen. Im Anschluss wird diese Summe durch die Anzahl der Beobachtungen n des entsprechenden Datensatzes geteilt (vgl. Pedregosa et al., 2011).

$$acc(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{\hat{y}^{(i)} = y^{(i)}\} \tag{22}$$

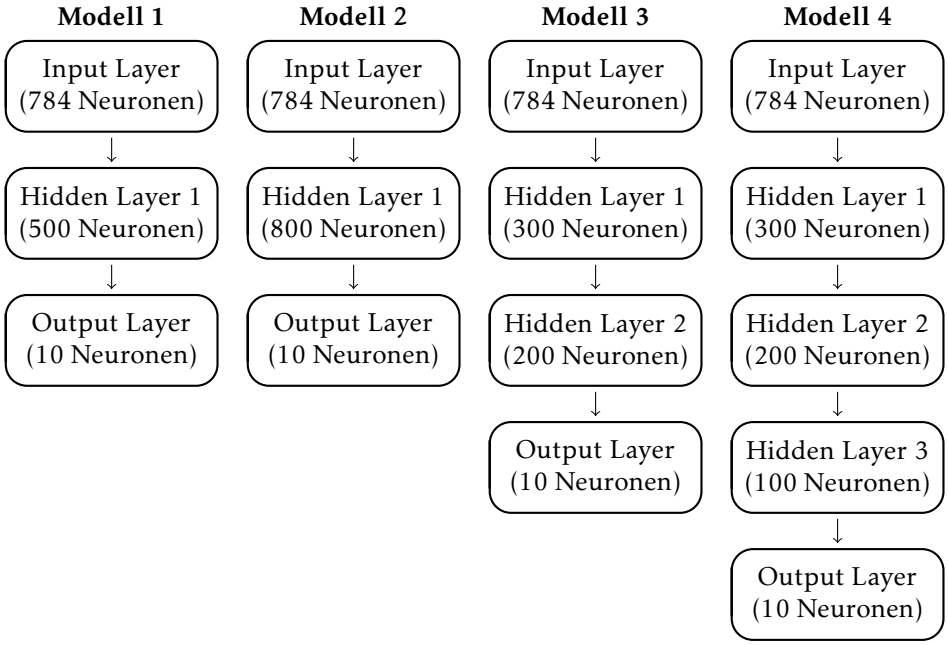
3.4.2 Anzahl der Hidden Layers und Neuronen

Zu Beginn der Hyperparametersuche werden unterschiedliche Netzwerkarchitekturen, d.h. Kombinationen aus Anzahl der Hidden Layers und Anzahl der darin enthaltenen Neuro-

[6] Häufig wird die Güte eines Modells auch durch dessen Fehlerrate angegeben, welcher folgendermaßen definiert ist: $error = 1 - acc$.

nen, betrachtet. Im Allgemeinen existieren dabei lediglich Heuristiken, an welchen man sich bei der Wahl einer Netzwerkarchitektur orientieren kann. In Abbildung 6 werden daher vier verschiedene Modellansätze dargestellt, wobei die Auswertung des Cross-Entropy Fehlers und der Performance am Validierungsdatensatz über 100 Epochen sich in Abbildung 7 befindet. Um den Vergleich zwischen den unterschiedlichen Modellausführungen zu ermöglichen, wird die Optimierung mit einer Lernrate von 0,3 und der logistischen Funktion als Aktivierungsfunktion der Hidden Layers durchgeführt.

Abbildung 6 Unterschiedliche Netzwerkarchitekturen von künstlichen neuronalen Netzen



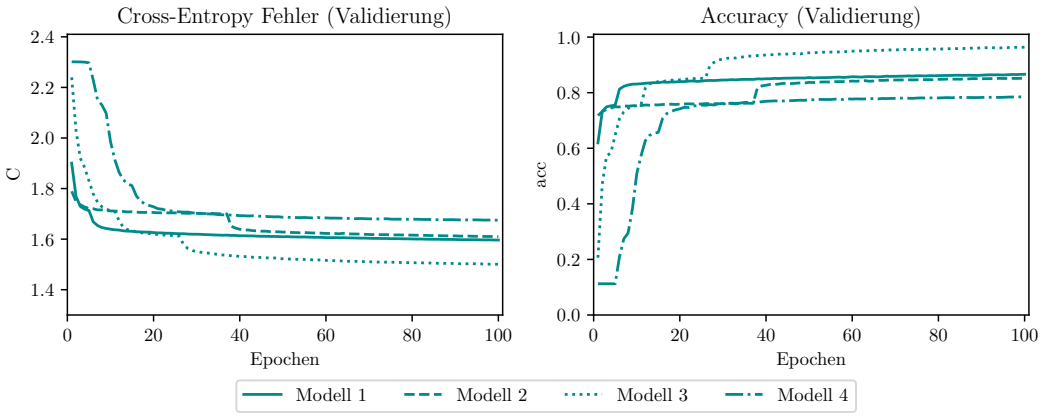
Des Weiteren kann das erste Modell, welches eine Hidden Layer mit 500 Neuronen besitzt, in den nachfolgenden Kapiteln 3.4.3 und 3.4.4 als Vergleichsmodell angesehen werden. Ein künstliches neuronales Netz mit einer Netzwerkarchitektur nach Modell 1 erreicht dabei letztendlich einen Accuracy Wert von 86,62% nach 100 Epochen. Das zweite Modell mit 800 Neuronen in der ersten Hidden Layer ist im Gegensatz dazu nur in der Lage 85,26% der Bilder des Validierungsdatensatzes korrekt zu klassifizieren. Dieses Ergebnis ist durch die Abwesenheit von Regularisierungsparameter[7] im Modellaufbau zu erklären. Grundsätzlich sollte nämlich die Anzahl der Hidden Neuronen eher zu hoch als zu niedrig angesetzt werden.

Modell 3 und 4 stellen dabei Deep Neural Networks dar, d.h. es handelt sich um Netzwerkarchitekturen mit mindestens zwei Hidden Layers. Außerdem besitzt die dritte Netzwerkarchitektur in Summe die gleiche Anzahl an Hidden Neuronen wie Modell 1. Jedoch erreicht das Modell 3 nach 100 Epochen einen Accuracy Wert von 96,35% am Validierungsdatensatz und Modell 1 lediglich einen Wert von 86,62%. Dieser Unterschied in der Güte der Modelle lässt sich dadurch erklären, dass durch das Hinzufügen weiterer Schichten in einem künstlichen neuronalen Netz dessen Fähigkeit neue Beobachtungen korrekt zu schätzen gesteigert wird, d.h. Modell 3 beinhaltet ein geringeres Risiko für *overfitting*.

[7] Regularisierungsparameter, wie z.B. *weight decay* in Verbindung mit *early stopping*, lassen überflüssige Gewichte bzw. Biaswerte gegen Null gehen, sodass diese keinen bzw. einen sehr geringen Einfluss auf die Schätzungen $\hat{y}$ ausüben. Der Einfachheit halber wird an dieser Stelle jedoch auf die Einstellung dieser zusätzlichen Hyperparameter verzichtet.

Der starke Performanceverlust in Modell 4 stellt hingegen eine Besonderheit bei der Verwendung der logistischen Funktion als Aktivierungsfunktion der Hidden Layers in Verbindung mit einer Gewichtsinitialiserung durch die Normalverteilung mit einem Mittelwert von 0 und einer Standardabweichung zwischen 0 und 1 in Deep Neural Networks dar. Dieses Problem wird durch sogenannte *vanishing gradients* beschrieben, welche bei der Durchführung von Backpropagation, d.h. bei der Rückführung der Fehler von der Output Layer zur Input Layer, entstehen. Dabei sind insbesondere die partiellen Ableitungen der Kostenfunktion nach den Gewichten und den Biaswerten der Schichten betroffen, die näher an der Input Layer liegen. Denn der rückführbare Fehler der Output Layer $\delta_j^{(L)}$ nimmt über die einzelnen Schichten hinweg ab. Letztendlich wird an den inputnäheren Schichten kein Update mehr durchgeführt, da die partiellen Ableitungen gegen null gehen. Somit konvergiert das Optimierungsverfahren Gradient Descent gegen eine sub-optimale Lösung. Im Falle von Modell 4 wird also lediglich ein Accuracy Wert von 78,40% nach 100 Epochen erreicht (vgl. Géron, 2017, S.273-278).

Abbildung 7 Cross-Entropy Fehler und Accuracy des Validierungsdatensatzes über 100 Epochen bei unterschiedlichen Netzwerkarchitekturen



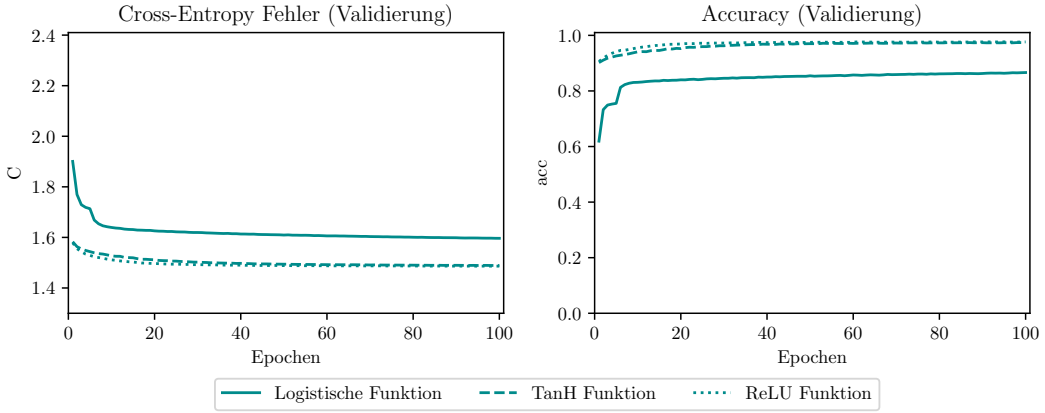
In Abbildung 7 ist zusätzlich der Cross-Entropy Fehler des Validierungsdatensatzes der vier Modelle über 100 Epochen abgebildet. Bei der Betrachtung der Fehlerwerte nach jeder Epoche ergibt sich bei allen Modellen eine stetige Reduktion des Cross-Entropy Fehlers.[8] Jedoch saturiert das Modell 1 bereits nach ungefähr 10 Epochen, während Modell 2 erst nach 40 Epochen saturiert. Modell 3 und 4 erreichen ein Minimum der Kostenfunktion nach ungefähr 30 Epochen. Zu beachten ist, dass aufgrund der nicht konvexen Kostenfunktion eine Vielzahl an lokalen Minima existieren. Prinzipiell besteht somit die Möglichkeit, dass auch nach 100 Epochen das globale Minimum nicht erreicht wird. Ein zusätzlicher Einflussfaktor auf das Endergebnis eines künstlichen neuronalen Netzes bzw. das Erreichen von Konvergenz gegen ein globales Minimum ist daher die zufällige Komponente bei der Initialisierung der Gewichte. Theoretisch müsste daher die Auswirkung mehrerer *random seeds* auf die Konvergenz des Learning Algorithmus getestet werden (vgl. Hastie et al., 2017, S.400 f.).

[8] Würde man den Cross-Entropy Fehler gegen die Iterationen und nicht gegen die Epochen abtragen, würde man weiterhin eine Reduktion der Kostenfunktion beobachten. Diese wäre jedoch nicht konsequent, sondern schwankend für jeden Mini-Batch.

3.4.3 Aktivierungsfunktionen

Die Aktivierungsfunktionen der Hidden Layers stellen einen weiteren Hyperparameter eines künstlichen neuronalen Netzes dar, welcher theoretisch für jeden Schicht individuell definiert werden kann. Jedoch wird an dieser Stelle lediglich eine einzige Aktivierungsfunktion für alle Hidden Layers ausgewählt. Der Vergleich zwischen der Sigmoid-, TanH- und ReLU-Funktion erfolgt dabei auf Basis des Modells 1 aus Kapitel 3.4.2, d.h. einer Netzwerkarchitektur mit 500 Neuronen in der ersten Hidden Layer und der Softmax Funktion als Aktivierungsfunktion der Neuronen der Output Layer.

Abbildung 8 Cross-Entropy Fehler und Accuracy des Validierungsdatensatzes über 100 Epochen bei unterschiedlichen Aktivierungsfunktionen der Hidden Layers



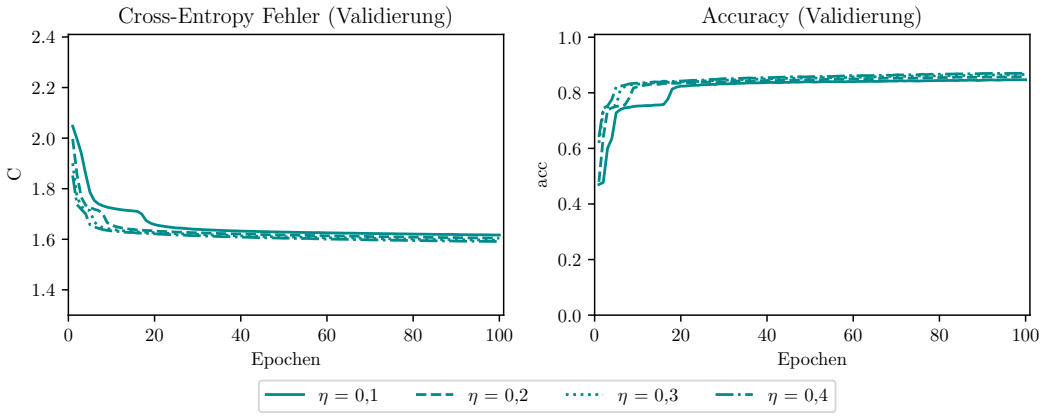
Konkret geht aus Abbildung 8 eine wesentliche Performanesteigerung des Gütemaßes Accuracy am Validierungsdatensatz durch die Verwendung der TanH- oder ReLU-Funktion gegenüber der logistischen Funktion in den Neuronen der Hidden Layer hervor. Denn das Modell 1 erreicht, wenn die logistische Funktion als Aktivierungsfunktion eingesetzt wird, lediglich einen Accuracy Wert von 86,62% nach 100 Epochen. Im Gegensatz dazu erzielt das künstliche neuronale Netz durch die Verwendung der TanH-Funktion als Aktivierungsfunktion der Hidden Layer einen Wert von 97,39%, wobei bei der Implementierung der ReLU-Funktion sogar 97,64% der Bilder des MNIST Validierungsdatensatzes nach 100 Epochen richtig klassifiziert werden können.

Obwohl im Falle der Netzwerkarchitektur von Modell 1 zwischen der Verwendung der TanH- und ReLU-Funktion eine geringe Differenz bezüglich der Güte liegt, hat sich die Verwendung der ReLU-Funktion als Aktivierungsfunktion der Hidden Layers als Standard im Bereich der künstlichen neuronalen Netze etabliert. Begründet wird die Präferenz der ReLU-Funktion durch die Beschleunigung der Konvergenz des Optimierungsverfahrens Mini-Batch Gradient Descent und der allgemeinen Berechnungsdauer eines künstlichen neuronalen Netzes (vgl. Karpathy, 2017a).

3.4.4 Lernrate

Als letzten Hyperparameter wird in dieser Arbeit die Lernrate η des Optimierungsverfahrens Mini-Batch Gradient Descent betrachtet, welche die Stärke der Gewichtsaktualisierungen beeinflusst. Im Allgemeinen eignet sich bei standardisierten oder skalierten Inputs eine Lernrate zwischen 10^{-6} und 1, wobei diese Empfehlung prinzipiell von der Parametrisierung des jeweiligen künstlichen neuronalen Netzes abhängt. Auf Basis dieser Heuristik werden daher vier verschiedene Lernraten innerhalb dieses Bereichs am Modell 1 aus Kapitel 3.4.2 unter gleichbleibender, restlicher Hyperparameter getestet (vgl. Bengio, 2012, S.8 f.).

Abbildung 9 Cross-Entropy Fehler und Accuracy des Validierungsdatensatzes über 100 Epochen bei unterschiedlichen Lernraten



Dabei kann aus Abbildung 9 eine beständige Abnahme des Cross-Entropy Fehlers über alle vier Lernraten festgestellt werden, welche nahezu exponentiell verläuft. Diese exponentielle Abnahme ist jedoch zudem ein Anzeichen für die Wahl einer geringfügig zu hohen Lernrate. Grundsätzlich verringern höhere Lernraten nämlich die Werte der Kostenfunktion schneller, da somit größere Updates an den Gewichten und den Biaswerten möglich sind. Demgegenüber erreichen sie aber lediglich sub-optimale Lösungen bezüglich der Werte der Kostenfunktion (vgl. Karpathy, 2017b).

Konkret erreicht eine Lernrate von 0,1 einen Accuracy Wert von 84,69% am Validierungsdatensatz nach 100 Epochen. Die weiteren Lernraten erzielen dabei konsequente Verbesserungen hinsichtlich der Performance, obwohl der Cross-Entropy Fehler darauf schließen lässt, dass der betrachtete Bereich an Lernraten zu hoch angesetzt wurde. So erzielt eine Lernrate von 0,2 einen Accuracy Wert von 85,90%, eine Lernrate von 0,3 einen Accuracy Wert von 86,62% und eine Lernrate von 0,4 einen Accuracy Wert von 87,09%. Die Dauer bis zum Erreichen des Plateaus variiert entsprechend der Größe der Lernrate, sodass der Learning Algorithmus Mini-Batch Gradient Descent bei einer Lernrate von 0,1 erst nach 25 Epochen gegen ein Minimum des Cross-Entropy Fehlers konvergiert. Dahingegen erfolgt die Konvergenz bei einer Lernrate von 0,4 bereits nach 10 Epochen.

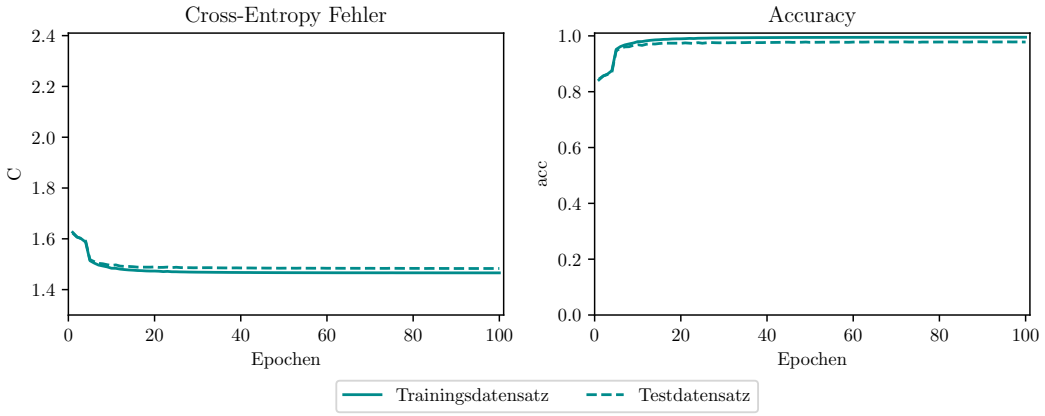
3.5 Evaluierung der Performance eines finalen Modells am Testdatensatz

Auf Grundlage der Ergebnisse der Hyperparametersuche in Kapitel 3.4, welche in Tabelle 3 zusammengefasst werden, ergibt sich das finale Modelle aus folgenden Hyperparametern: einer Netzwerkarchitektur des Modells 3, d.h. ein dreischichtiges künstliches neuronales Netz mit 300 Neuronen in der ersten Hidden Layer und 200 Neuronen in der zweiten Hidden Layer, einer Verwendung der ReLU Aktivierungsfunktion in den Hidden Layers und einer Lernrate von 0,4. Die Auswahl eines Hyperparameters erfolgt somit basierend auf der höchsten Accuracy des Validierungsdatensatzes unter den jeweiligen Modellen, welche in Tabelle 3 mit * gekennzeichnet wurden. Die restlichen Hyperparameter sind weiterhin gemäß der Definition in Kapitel 3.4 zu verwenden.

Tabelle 3 Accuracy des Validierungsdatensatzes von Modellen mit unterschiedlichen Hyperparametern

Netzwerkarchitektur		Aktivierungsfunktion		Lernrate	
Modell 1	86,62%	Sigmoid	86,62%	$\eta = 0,1$	84,69%
Modell 2	85,26%	TanH	97,39%	$\eta = 0,2$	85,90%
Modell 3*	96,35%	ReLU*	97,64%	$\eta = 0,3$	86,62%
Modell 4	78,40%			$\eta = 0,4$*	87,09%

Abbildung 10 Cross-Entropy Fehler und Accuracy des endgültigen Modells am Trainings- und Testdatensatz über 100 Epochen



Die Performance des finalen Modells mit obiger Konfiguration der Hyperparameter wird im Anschluss nicht am Validierungs- sondern am Testdatensatz evaluiert. Aus Abbildung 10 geht dabei eine geringfügige Diskrepanz bezüglich des Cross-Entropy Fehlers zwischen dem Trainings- und dem Testdatensatz hervor, wodurch lediglich eine geringes Maß an *overfitting* besteht. Das auf Basis des Trainingsdatensatzes erstellte Modell lässt sich somit auf neue Beobachtungen anwenden und erreicht einen Accuracy Wert von 97,80% am Testdatensatz, welcher höher ist als die zuvor getesteten Modelle aus Tabelle 3. Der *test error* beträgt also lediglich 2,2%, womit die Güte des endgültigen Modells in dieser Arbeit mit den erzielten Fehlerraten aus dem Paper von Lecun et al. (1998) vergleichbar ist, welcher im Bereich der künstlichen neuronalen Netze lediglich über sogenannte *distortions*

und Convolutional Neural Networks eine bessere Performance erlangt hat. Die geringste veröffentlichte Fehlerrate des MNIST Datensatzes beträgt dabei momentan 0,18% unter Verwendung von Random Multimodel Deep Learning Modellen (vgl. Kowsar et al., 2018).

4 Ausblick

Die in dieser Arbeit betrachteten Feedforward Neural Networks stellen lediglich die Grundlage für zahlreiche weitere Typen von künstlichen neuronalen Netzen dar. So kann beispielsweise die Performance von Bilderkennungsverfahren, wie im Falle der Analyse des MNIST Datensatzes aus handgeschriebener Zahlen, durch die Verwendung von Convolutional Neural Networks gesteigert werden. Im Wesentlichen reduziert dabei diese Art von künstlichen neuronalen Netzen die Anzahl der zu optimierenden Parameter innerhalb eines Netzwerks durch die Anwendung sogenannter Convolutional und Pooling Layers. Im Gegensatz zu den vollständig verbundenen Schichten eines Feedforward Neural Networks verbinden dabei die Convolutional Layers z.B. die Neuronen der Input Layer nur noch zum Teil mit den Neuronen der ersten Hidden Layer. Diese zusätzlichen Schichten beschleunigen nicht nur die Berechnung eines künstlichen neuronalen Netzes, sondern verringern zudem das Risiko von *overfitting* durch eine niedrigere Anzahl an Gewichten.[9]

Neben Convolutional Neural Networks existieren außerdem Recurrent Neural Networks, welche sich insbesondere zur Analyse von sequentiellen Daten eignen. Konkrete Anwendungsgebiete dieser Art von künstlichen neuronalen Netzen sind somit beispielsweise die Vorhersage von Aktienkursen oder die Analyse von Text, Sprache oder Ton im Bereich des Natural Language Processing. Im Allgemeinen stellen Recurrent Neural Networks eine Erweiterung von Feedforward Neural Networks dar, indem sie zusätzliche Verbindungen zu Neuronen vorheriger Schichten zulassen. Diese Verbindungen sind wiederum als Gewichte zu interpretieren, welche mit den Outputs eines vorherigen Zeitpunkts multipliziert werden (vgl. Géron, 2017, S.359-361, S.385-389).

Grundsätzlich eignen sich künstliche neuronale Netze somit für eine Vielzahl von Anwendungsbereichen, wobei die Wahl zwischen einem Feedforward, Convolutional oder Recurrent Neural Network bei unterschiedlichen Formen von Inputs die Performance eines Modells maßgeblich beeinflusst. Letztendlich stellt der Typ eines künstlichen neuronalen Netzes daher einen weiteren Hyperparameter dar, welcher durch Heuristiken und Hyperparameteroptimierung eingestellt werden muss, um die Performance eines künstlichen neuronalen Netzes zu verbessern.

[9] Diese Vorteile machen sich vor Allem bei größeren Bildern bemerkbar, welche z.B. bei einer Größe von 100 x 100 x 3 bereits 30.000 Input Variablen und somit Gewichte zu einem einzigen Neuron in der ersten Hidden Layer eines elementaren Feedforward Neural Networks besitzen würden. Im Gegensatz dazu besitzen die MNIST Bilder nämlich lediglich eine Größe von 28 x 28 x 1 und daher 784 Gewichte zu einem Neuron der ersten Hidden Layer.

A Anwendung des Backpropagation Algorithmus

Die Formeln in Kapitel 2.2.3 über den Backpropagation Algorithmus sollen an dieser Stelle an einem konkreten Beispiel hergeleitet bzw. angewandt werden. Das Beispiel besteht dabei aus einem Klassifizierungsproblem, wobei der Cross-Entropy Fehler aus Gleichung (9) als Kostenfunktion ausgewählt wird. Außerdem wird die logistische Funktion aus Gleichung (5) als Aktivierungsfunktion für alle Hidden Layers und die Output Layer verwendet (vgl. Sadowski, 2016, S.1 f.).

Im Folgenden wird die Berechnung der partiellen Ableitung der Cross-Entropy Kostenfunktion nach dem Gewicht $w_{jk}^{(L)}$ und $w_{jk}^{(L-1)}$ durchgeführt. Im ersten Fall handelt es sich also um das Gewicht $w_{jk}^{(L)}$ zwischen der letzten Hidden Layer $(L-1)$ und der Output Layer (L), wohingegen im zweiten Fall ein Gewicht der Schicht (l) betrachtet wird, wobei es sich konkret um das Gewicht $w_{jk}^{(L-1)}$ zwischen der Hidden Layer $(L-2)$ und $(L-1)$ handelt.

1) Berechnung der partiellen Ableitung $\frac{\partial C}{\partial w_{jk}^{(L)}}$

$$\frac{\partial C}{\partial w_{jk}^{(L)}} = \frac{\partial C}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} = \delta_j^{(L)} a_k^{(L-1)} \tag{23}$$

Gleichung (23) beinhaltet drei partielle Ableitungen, welche in den folgenden Gleichungen separat berechnet werden:

$$\frac{\partial C}{\partial \hat{y}_j} = \frac{\partial}{\partial \hat{y}_j} -(y_j \log(\hat{y}_j) + (1-y_j) \log(1-\hat{y}_j)) = -\frac{y_j}{\hat{y}_j} + \frac{(1-y_j)}{(1-\hat{y}_j)} = \frac{\hat{y}_j - y_j}{\hat{y}_j(1-\hat{y}_j)} \tag{24}$$

$$\frac{\partial \hat{y}_j}{\partial z_j^{(L)}} = \frac{\partial}{\partial z_j^{(L)}} \sigma(z_j^{(L)}) = \sigma'(z_j^{(L)}) = \sigma(z_j^{(L)})\,(1-\sigma(z_j^{(L)})) = \hat{y}_j(1-\hat{y}_j) \tag{25}$$

$$\frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} = \frac{\partial}{\partial w_{jk}^{(L)}} \sum_k w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)} = a_k^{(L-1)} \tag{26}$$

Die Ableitung der logistischen Funktion in Gleichung (25) hat dabei die nützliche Eigenschaft, dass diese durch die Funktion selbst dargestellt werden kann. Anschließend kann man die Gleichungen (24) und (25) zusammenfügen und man erhält den Fehlerterm $\delta_j^{(L)}$ der Output Layer (L).

$$\delta_j^{(L)} = \frac{\partial C}{\partial z_j^{(L)}} = \frac{\hat{y}_j - y_j}{\hat{y}_j(1-\hat{y}_j)} \hat{y}_j(1-\hat{y}_j) = (\hat{y}_j - y_j) \tag{27}$$

Der ermittelte Fehlerterm aus Gleichung (27) und der Term aus Gleichung (26) wird nun in die anfängliche Gleichung (23) eingesetzt und es ergibt sich die gesuchte par-

tielle Ableitung der Kostenfunktion nach dem Gewicht $w_{jk}^{(L)}$.

$$\frac{\partial C}{\partial w_{jk}^{(L)}} = \underbrace{\frac{(\hat{y}_j - y_j)}{\hat{y}_j(1-\hat{y}_j)}}_{\frac{\partial C}{\partial \hat{y}_j}} \underbrace{\hat{y}_j(1-\hat{y}_j)}_{\frac{\partial \hat{y}_j}{\partial z_j^{(L)}}} \underbrace{a_k^{(L-1)}}_{\frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}}} = \underbrace{(\hat{y}_j - y_j)}_{\frac{\partial C}{\partial z_j^{(L)}}} a_k^{(L-1)} \tag{28}$$

2) Berechnung der partiellen Ableitung $\frac{\partial C}{\partial w_{jk}^{(L-1)}}$

$$\frac{\partial C}{\partial w_{jk}^{(L-1)}} = \delta_j^{(L-1)} a_k^{(L-2)} = \left(\sum_j \delta_j^{(L)} w_{jk}^{(L)} \right) f'(z_j^{(L-1)}) a_k^{(L-2)} \tag{29}$$

Die konkrete Formulierung von Gleichung (19) wird in Gleichung (29) dargestellt und in Gleichung (30) und (31) ausgewertet.

$$\frac{\partial C}{\partial w_{jk}^{(L-1)}} = \left(\sum_j \underbrace{\frac{\partial C}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial z_j^{(L)}}}_{} \underbrace{\frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}}}_{} \right) \underbrace{\frac{\partial a_k^{(L-1)}}{\partial z_k^{(L-1)}}}_{} \underbrace{\frac{\partial z_k^{(L-1)}}{\partial w_{jk}^{(L-1)}}}_{} \tag{30}$$

$$= \left(\sum_j (\hat{y}_j - y_j) \quad w_{jk}^{(L)} \right) \sigma'(z_k^{(L-1)}) a_k^{(L-2)} \tag{31}$$

B Backpropagation Algorithmus in Matrixnotation

1) Berechnung des Fehlertermvektor $\boldsymbol{\delta}^{(L)}$ der Output Layer (L)

$$\boldsymbol{\delta}^{(L)} = \nabla_{\boldsymbol{z}^{(L)}} C = \nabla_{\hat{\boldsymbol{y}}} C \odot f'(\boldsymbol{z}^{(L)}) \tag{32}$$

Gleichung (32) stellt somit die Verallgemeinerung der Gleichung (17) dar und beinhaltet sämtliche Fehlerterme $\delta_j^{(L)}$. Anzumerken ist, dass es sich bei dem Operator $\odot$, um das elementweise Produkt zweier Matrizen mit gleichen Dimensionen handelt. Des Weiteren erfolgt die Ableitung $f'(\boldsymbol{z}^{(L)})$ ebenfalls elementweise.

2) Berechnung des Fehlertermvektor $\boldsymbol{\delta}^{(l)}$ der Schicht (l)

$$\boldsymbol{\delta}^{(l)} = (\boldsymbol{W}^{(l+1)\top} \boldsymbol{\delta}^{(l+1)}) \odot f'(\boldsymbol{z}^{(l)}) \tag{33}$$

Die Verallgemeinerung der Gleichung (20) erfolgt durch Gleichung (33). Die Gewichtsmatrix wird transponiert, da bei der Anwendung des Backpropagation Algorithmus die Summe über den Index j läuft, d.h. über die Neuronen der Schicht $(l+1)$. Im Gegensatz dazu läuft die Summe bei Forward Propagation über den Index k und somit über die Neuronen der Schicht $(l-1)$.

3) Berechnung der Gradientenmatrix $\nabla_{\boldsymbol{W}^{(l)}} C$

$$\nabla_{\boldsymbol{W}^{(l)}} C = \boldsymbol{\delta}^{(l)} \boldsymbol{a}^{(l-1)\top} \tag{34}$$

Die Matrix in Gleichung (34) beinhaltet sämtliche individuellen partiellen Ableitungen, welche in Gleichung (19) definiert wurden.

4) Berechnung der Gradientenvektors $\nabla_{\boldsymbol{b}^{(l)}} C$

$$\nabla_{\boldsymbol{b}^{(l)}} C = \boldsymbol{\delta}^{(l)} \tag{35}$$

Der Vektor in Gleichung (35) beinhaltet sämtliche individuellen partiellen Ableitungen, welche in Gleichung (21) definiert wurden bzw. entspricht dem Vektor in Gleichung (33) (vgl. Ng, 2017a, S.2).

Symbolverzeichnis

	Indizes und Superskripte
n	Anzahl der Beobachtungen im Trainingsdatensatz
n_m	Anzahl der Beobachtungen in einem Mini-Batch
m	Anzahl der Iterationen
i	Beobachtung (i) aus dem Trainingsdatensatz
n_x	Anzahl der Input-Variablen
P	Anzahl der der Output-Variablen bzw. Klassen
L	Anzahl der Schichten im künstlichen neuronalen Netz
l	Schicht (l) innerhalb des künstlichen neuronalen Netzes
k	vom Input Neuron weggehend
j	zum Output Neuron hingehend
	Objekte
x_k	Inputs aus dem Trainingsdatensatz
y_j	Targets bzw. Sollwerte aus dem Trainingsdatensatz
$\hat{y}_j$	geschätzte Output Werte
$w_{jk}^{(l)}$	Gewicht vom Neuron k (Input Neuron) in der Schicht $(l-1)$ zum Neuron j (Output Neuron) in der Schicht (l)
$b_j^{(l)}$	Bias vom Neuron j in der Schicht (l)
$z_j^{(l)}$	Summe der gewichten Inputs inklusive Bias des Neuron j in der Schicht (l)
$a_j^{(l)}$	Output des Neurons j in der Schicht (l)
	Matrizen und Vektoren
$\boldsymbol{x}$	Input Vektor aus dem Trainingsdatensatz
$\boldsymbol{y}$	Target Vektor aus dem Trainingsdatensatz
$\hat{\boldsymbol{y}}$	geschätzter Output Vektor
$\boldsymbol{W}^{(l)}$	Gewichtsmatrix der Schicht (l)
$\boldsymbol{b}^{(l)}$	Bias Vektor der Schicht (l)
$\boldsymbol{z}^{(l)}$	gewichteter Input Vektor der Schicht (l)
$\boldsymbol{a}^{(l)}$	Output Vektor der Schicht (l)
	Funktionen und Operatoren
$f(\cdot)$	Aktivierungsfunktion
$C(\cdot)$	Kostenfunktion
∇	Gradienten Operator
$\odot$	elementweises Produkt von Matrizen gleicher Größe
	Sonstiges
η	Lernrate
Θ	Schwellenwert

Abkürzungsverzeichnis

vgl.	vergleiche
S.	Seite
d.h.	das heißt
z.B.	zum Beispiel
bzw.	beziehungsweise

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu und X. Zheng (2016). TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, S. 265–283.

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. *CoRR*.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

Bishop, C. M. (2009). *Pattern Recognition and Machine Learning*. Springer.

Dering, M. L. und C. S. Tucker (2017). A Convolutional Neural Network Model for Predicting a Product's Function, Given Its Form. *Journal of Mechanical Design - Transactions of the ASME 139*.

Goodfellow, I., Y. Bengio und A. Courville (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Géron, A. (2017). *Hands-On Machine Learning with Scikit-Learn & TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly.

Hastie, T., R. Tibshirani und J. Friedman (2017). *The Elements of Statistical Learning*. Springer.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation*. Prentice Hall.

Karpathy, A. (2017a). *Neural Networks Part 1: Setting up the Architecture*. http://cs231n.github.io/neural-networks-1. Stand: 22.05.2018.

Karpathy, A. (2017b). *Neural Networks Part 3: Learning and Evaluation*. http://cs231n.github.io/neural-networks-3. Stand: 27.05.2018.

Kowsar, K., M. Heidarysafa, D. E. Brown, K. J. Meimandi und L. E. Barnes (2018). RMDL: Random Multimodel Deep Learning for Classification. *Proceedings of the 2018 International Conference on Information System and Data Mining*.

Lantz, B. (2013). *Machine Learning with R*. Packt Publishing.

Lecun, Y., L. Bottou, Y. Bengio und P. Haffner (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, S. 2278–2324.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

Ng, A. (2017a). *CS229: Additional Notes on Backpropagation*. http://cs229.stanford.edu/notes/cs229-notes-backprop.pdf. Stand: 05.05.18.

Ng, A. (2017b). *CS229 Lecture Notes Deep Learning*. http://cs229.stanford.edu/notes/cs229-notes-deep_learning.pdf. Stand: 05.05.18.

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press. http://neuralnetworksanddeeplearning.com/.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot und E. Duchesnay (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research 12*, 2825–2830.

Rashid, T. (2017). *Neuronale Netze selbst programmieren: Ein verständlicher Einstieg mit Python*. O'Reilly.

Sadowski, P. (2016). *Notes on Backpropagation*. https://www.ics.uci.edu/~pjsadows/notes.pdf. Stand: 10.05.18.

Schriftliche Versicherung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Weiterhin wurde diese Arbeit keiner anderen Prüfungsbehörde vorgelegt.

Regensburg, den 08. Juni 2018